RTI Message Service

User's Manual

Version 4.5

The Global Leader in DDS

Printed in U.S.A. First printing.
Oct. 2011.

Trademarks

Copy and Use Restrictions

Third-Party Copyright Notices

Note: In this section, "the Software" refers to third-party software, portions of which are used in *RTI Message Service*; "the Software" does not refer to *RTI Message Service*.

- Portions of this product were developed using MD5 from Aladdin Enterprises.
- Portions of this product include software derived from Fnmatch, (c) 1989, 1993, 1994 The Regents of the University of California. All rights reserved. The Regents and contributors provide this software "as is" without warranty.
- Portions of this product were developed using EXPAT from Thai Open Source Software Center Ltd and Clark Cooper Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd and Clark Cooper Copyright (c) 2001, 2002 Expat maintainers. Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions: The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

Technical Support

Real-Time Innovations, Inc.
385 Moffett Park Drive
Sunnyvale, CA 94089
Phone: (408) 990-7444
Email: support@rti.com
Website: https://support.rti.com/

Contents

Part 2:Advanced Concepts

6 Scalable High-Performance Applications: Message Reliability

7 Scalable High-Performance Applications: Durability and Persistence for High Availability

Chapter 1 Welcome to RTI Message Service

Welcome to *RTI® Message Service*, the highest-performing JMS-compliant messaging system in the world. *RTI Message Service* makes it easy to develop, deploy and maintain distributed applications. Its core messaging technology has been proven in hundreds of unique designs for life- and mission-critical applications across a variety of industries, providing

- ❏ ultra-low latency and extremely high throughput
- ❏ with industry-leading latency determinism
- ❏ across heterogeneous systems spanning thousands of applications.

Its extensive set of real-time quality-of-service parameters allows you to fine-tune your application to meet a wide range of timeliness, reliability, fault-tolerance, and resource usage-related goals.

This chapter introduces the basic concepts within the middleware and summarizes how *RTI Message Service* addresses the needs of high-performance systems. It also describes the documentation resources available to you and provides a road map for navigating them. Specifically, this chapter includes:

1.1 Benefits of RTI Message Service

RTI Message Service is publish/subscribe networking middleware for high-performance distributed applications. It implements the Java Message Service (JMS) specification, but it is not just another MOM (message-oriented middleware). Its unique peer-to-peer architecture and targeted high-performance and real-time capabilities extend the specification to provide unmatched value.

1.1.1 Reduced Risk Through Industry-Leading Performance and Availability

RTI Message Service provides industry-leading performance, whether measured in terms of latency, throughput, or real-time determinism. One contributor to this superior performance is RTI's unique architecture, which is entirely peer-to-peer.

Traditional messaging middleware implementations require dedicated servers to broker message flows, crippling application performance, increasing latency, and introducing time non-determinism. These brokers increase system administration costs and can represent single points of failure within a distributed application, putting data reliability and availability at risk.

RTI eliminates broker overhead by allowing messages to flow directly from a publisher to each of its subscribers in a strictly peer-to-peer fashion. At the same time, it provides a variety of powerful capabilities to ensure high availability.

Traditional message-oriented middleware implementations require a broker to forward every message, increasing latency and decreasing determinism and fault tolerance. RTI's unique peer-to-peer architecture eliminates bottlenecks and single points of failure.

Redundancy and high availability can optionally be layered onto the peer-to-peer data fabric by transparently inserting instances of *RTI Persistence Service*. These instances can distribute the load across topics and can also be arbitrarily redundant to provide the level of data availability your application requires. See Chapter 7, "Scalable High-Performance Applications: Durability and Persistence for High Availability," in the User's Manual for more information about this capability.

Publishers and subscribers can enter and leave the network at any time, and the middleware will connect and disconnect them automatically. *RTI Message Service* provides fine-grained control over fail-over among publishers, as well as detailed status notifications to allow applications to detect missed delivery deadlines, dropped connections, and other potential failure conditions. See Chapter 6, "Fault Tolerance," in the Configuration and Operation Manual for more information about these capabilities.

1.1.2 Reduced Cost through Ease of Use and Simplified Deployment

- **Increased developer productivity**—Easy-to-use, well-understood JMS APIs get developers productive quickly. (Take an opportunity to go through the tutorial in the *Getting Started Guide* if you haven't already.) Outside of the product documentation itself, a wide array of third-party JMS resources exist on the web and on the shelves of your local book store.
- **Simplified deployment**—Because *RTI Message Service* consists only of dynamic libraries, you don't need to configure or manage server machines or processes. That translates into faster turnaround and lower overhead for your team.
- **Reduced hardware costs**—Some traditional messaging products require you to purchase specialized acceleration hardware in order to achieve high performance. The extreme efficiency and reduced overhead of RTI's implementation, on the other hand, allows you to see strong performance even on commodity hardware.

1.1.3 Unmatched Power and Flexibility to Meet Unique Requirements

When you need it, RTI provides a high degree of fine-grained, low-level control over the operation of the middleware, including, but not limited to:

- The volume of meta-traffic sent to assure reliability.
- The frequencies and timeouts associated with all events within the middleware.
- The amount of memory consumed, including the policies under which additional memory may be allocated by the middleware.

These quality-of-service (QoS) policies can be specified in configuration files so that they can be tested and validated independently of the application logic. When they are not specified, the middleware will use default values chosen to provide good performance for a wide range of applications.

For specific information about the parameters available to you, consult the Configuration and Operation Manual.

1.1.4 Interoperability with OMG Data Distribution Service-Based Systems

The Data Distribution Service (DDS) specification from the Object Management Group (OMG) has become the standard for real-time data distribution and publish/subscribe messaging for high performance real-time systems, especially in the aerospace and defense industries. *RTI Message Service* is the only JMS implementation to directly interoperate at the wire-protocol level with *RTI Data Distribution Service*, the leading DDS implementation.

RTI Data Distribution Service is available not only in Java but also in several other managed and unmanaged languages. It is supported on a wide variety of platforms, including embedded hardware running real-time operating systems. For more information, consult your RTI account representative. If you are already an *RTI Data Distribution Service* user, and are interested in DDS/JMS interoperability, consult the Interoperability Guide that accompanies this documentation.

1.2 Features of RTI Message Service

Under the hood, *RTI Message Service* goes beyond the basic JMS publish-subscribe model to target the needs of applications with high-performance, real-time, and/or low-overhead requirements and provide the following:

- **Peer-to-peer publish-subscribe communications** Simplifies distributed application programming and provides time-critical data flow with minimal latency.
  - Clear semantics for managing multiple sources of the same data.
  - Efficient data transfer, customizable Quality of Service, and error notification.
  - Guaranteed periodic messages, with minimum and maximum rates set by subscriptions, including notifications when applications fail to meet their deadlines.
  - Synchronous or asynchronous message delivery to allow applications control over the degree of concurrency.
  - Ability to send the same message to multiple subscribers efficiently, including support for reliable multicast with customizable levels of positive and negative message acknowledgement.

- **Reliable messaging**—Enables subscribing applications to not only specify reliable delivery of messages, but to customize the degree of reliability required. Data flows can be configured for (1) guaranteed delivery at any cost, at one extreme, (2) the lowest possible latency and highest possible determinism, even if it means that some messages will be lost, at the other extreme, or (3) many points in between.
- **Multiple communication networks**—Multiple independent communication networks (*domains*), each using *RTI Message Service*, can be used over the same physical network to isolate unrelated systems and subsystems. Individual applications can be configured to participate in one or multiple domains.
- **Symmetric architecture**—Makes your application robust:
  - No central server or privileged nodes, so the system is robust to application and/or node failures.
  - Topics, subscriptions, and publications can be dynamically added and removed from the system at any time.

Multiple network transports—*RTI Message Service* includes support for UDP/IP (v4 and v6)—including, for example, Ethernet, wireless, and Infiniband networks—and shared memory transports. It also includes the ability to dynamically plug in support for additional network transports and route messages over them. It can optionally be configured to operate over a variety of transport mechanisms, including backplanes, switched fabrics, and other networking technologies.

Multi-platform and heterogeneous system support—Applications based on *RTI Message Service* can communicate transparently with each other regardless of the underlying operating system or hardware. Consult the Release Notes to see which platforms are supported in this release.

Vendor neutrality and standards compliance—The *RTI Message Service* API complies with the JMS specification. Unlike other JMS implementations, it also supports a wire protocol that is open and standards-based: the Real-Time Publish/Subscribe (RTPS) protocol specification from the Object Management Group (OMG), which extends the International Engineering Consortium's (IEC's) publicly available RTPS specification. This protocol also enables interoperability between *RTI Message Service* and *RTI Data Distribution Service* and between various DDS implementations. See Interoperability with OMG Data Distribution Service-Based Systems (Section 1.1.4).

1.3 JMS Conformance

RTI Message Service is a high-performance messaging platform for demanding applications, including applications with real-time requirements. Not all portions of the JMS specification are relevant or appropriate for this domain, and some required features are not included in the specification. For more information about JMS conformance, including both limitations and significant extensions, see Appendix A, "JMS Conformance," in the User's Manual.

1.4 Understanding and Navigating the Documentation

To get you from your download to running software as quickly as possible, we have divided this documentation into several parts.

- Release Notes—Provides system-level requirements and other platform-specific information about the product. *Those responsible for installing RTI Message Service should read this document first.*
- Getting Started Guide—Describes how to download and install *RTI Message Service*. It also lays out the core value and concepts behind the product and takes you step-by-step through the creation of a simple example application. *Developers should read this document first.*
- User's Manual—Describes the features of the product, their purpose and value, and how to use them. It is aimed at developers who are responsible for implementing the functional requirements of a distributed system, and is organized around the structure of the JMS APIs and certain common high-level scenarios.
- Configuration and Operation Manual—Provides lower-level, more in-depth configuration information and focuses on system-level concerns. It is aimed at engineers who are responsible for configuring, optimizing, and administering *RTI Message Service*-based distributed systems.

Many readers will also want to consult additional documentation available online. In particular, RTI recommends the following:

- **RTI Self-Service Portal**—http://www.rti.com/support. Select the **Find Solution** link to see sample code, general information on *RTI Message Service*, performance information, troubleshooting tips, and other technical details.

- **RTI Example Performance Test**—This recommended download includes example code and configuration files for testing and optimizing the performance of a simple *RTI Message Service*-based application on your system. The program will test both throughput and latency under a wide variety of middleware configurations. It also includes documentation on tuning the middleware and the underlying operating system.

  To download this test, first log into your self-service support portal as described above. Click **Find Solution** in the menu bar at the top of the page then click **Performance** under **All Solutions** in the resulting page. Finally, click on or search for **Example Performance Test** to download the test.

  You can also review the data from several performance benchmarks here: http://www.rti.com/products/jms/latency-throughput-benchmarks.html.

- **Java Message Service (JMS) API Documentation**—*RTI Message Service* APIs are compliant with the JMS specification. This specification is a part of the broader Java Enterprise Edition (Java EE) product from Sun Microsystems; Java EE 5 is documented at http://java.sun.com/javaee/5/docs/api/. In particular, see the **javax.jms** package.

- **Java Standard Edition API Documentation**—Java EE is an extension to, and relies on types imported from, the Java Standard Edition (Java SE) product. Java SE 6 is documented online at http://java.sun.com/javase/6/docs/api/.

- **Whitepapers and other articles** are available from http://www.rti.com/resources/.

Part 1:Core Concepts

This part of the manual includes the following chapters:

Chapter 2 Connecting to the Network

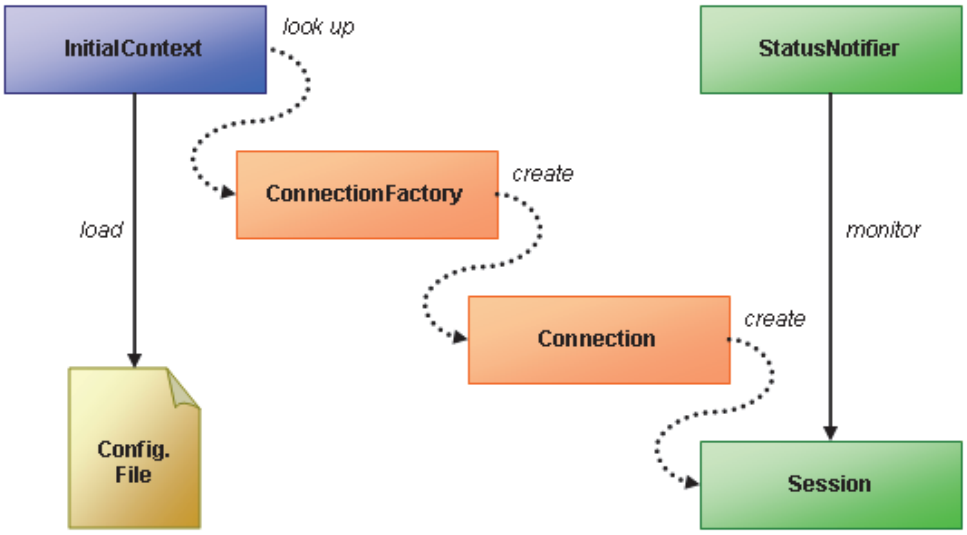
This chapter describes the basic concepts you need in order to connect to the network, including how to look up administered objects in XML configuration files and use them to begin communication. It will take you through the following steps:

1. Create a configuration file and define a *Topic* and *ConnectionFactory* in it.
2. Create an *InitialContext* and use it to look up the *ConnectionFactory* you defined.
3. Use the *ConnectionFactory* to create a *Connection*.
4. Use the *Connection* to create a *Session*.
5. Use a *StatusNotifier* attached to that *Session* to obtain synchronous or asynchronous notifications about status changes related to these objects.

The steps above are the same for publishing and subscribing applications. The subsequent chapters in this manual—Chapter 3: Messages and Topics, Chapter 4: Publishing Messages, and Chapter 5: Subscribing to Messages—will pick up where this chapter leaves off.

You should also read the Getting Started Guide, and go through the tutorial in that document before reading the more in-depth information in this manual.

The objects described in this chapter have the following relationships:



This chapter is organized as follows:

- Configuring the Middleware (Section 2.1)
- Beginning Communication (Section 2.2)
- Responding to Network Events (Section 2.3)
- Introduction to Peer-to-Peer Discovery (Section 2.4)

2.1 Configuring the Middleware

The object model of *RTI Message Service* middleware is grounded in the administered objects identified by the JMS specification:

- **Topic**: Topics identify logical destinations to which message producers publish, and from which message consumers subscribe to, messages. Topics are named, and they configure the qualities of service (QoS) of associated producers and consumers. Topics are described in more detail in Chapter 3: Messages and Topics.

- **ConnectionFactory**: A connection factory is the root of, and factory for, a hierarchy of sessions, message producers, and message consumers. Nearly all objects in the API are created, directly or indirectly, from a connection factory. Connection factories are described in detail later in this chapter.

2.1.1 File-Based Configuration Format

RTI Message Service is primarily configured using files in an XML-based format. That format is documented in detail in the Configuration and Operation Manual but is summarized here and in subsequent sections of this manual. Many application developers will not need to consult the Configuration and Operation Manual.

The configuration file is rooted in the element **<jms></jms>**.

2.1.1.1 Libraries

All administered objects are organized into named groups called *libraries*. Libraries allow related objects to be grouped together and to be segmented from unrelated objects.

A library is declared with the element **<library></library>**, which has a single attribute: **name**. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<jms>
    <library name="Example">
            <!-- ... -->
    </library>
</jms>
```

2.1.1.2 Topics

Topics are declared within libraries with the element **<topic></topic>**, which has a single attribute: **name**. For example:

```
<library name="Example">
    <topic name="Example Topic">
            <!-- ... -->
    </topic>
</library>
```

More information about topics can be found in the chapter Messages and Topics.

2.1.1.3 Connection Factories

Connection factories are declared within libraries with the element **<connection_factory></connection_factory>**, which has a single attribute: **name**. For example:

```
<library name="Example">
    <connection_factory name="Example Connection Factory">
            <!-- ... -->
    </connection_factory>
</library>
```

More information about connection factories can be found in Beginning Communication (Section 2.2).

2.1.2 Loading a Configuration File

Interface: **javax.naming.Context**

Interface: **com.rti.naming.RTIContext extends javax.naming.Context**

Class: **javax.naming.InitialContext implements javax.naming.Context**

Class: **com.rti.naming.InitialContext extends javax.naming.InitialContext implements RTIContext**

Administered objects are loaded using an "initial context" of type **javax.naming.InitialContext or com.rti.naming.InitialContext**, which parses a file and looks up the administered objects defined therein. When your application is ready to begin communication, it will first load a *ConnectionFactory* (see Connection Factory (Section 2.2.1)) from the context.

2.1.2.1 Creating an InitialContext

Constructor: **com.rti.naming.InitialContext(java.util.Map<?, ?> environment) throws javax.naming.NamingException**

Constructor: **javax.naming.InitialContext(java.util.Hashtable<?, ?> environment) throws javax.naming.NamingException**

The initial context must be able to load the code that can parse the *RTI Message Service* configuration file. To do this, its environment properties must contain at least two properties:

- **javax.naming.Context.INITIAL_CONTEXT_FACTORY**, set to the name of the **com.rti.jms.JmsConfigContextFactory** class. This class initiates the parsing of the configuration file. If you use the RTI *InitialContext* implementation instead of its **javax.naming** superclass, this property will be set automatically.
- A URL identifying the location of the configuration file. This URL is identified by the key **javax.naming.Context.PROVIDER_URL**. While the URL is conventionally specified in string form, *RTI Message Service* will also accept a **java.net.URL** or **java.io.File** object. The file path can be relative or absolute.

The constructor will throw a *NamingException* if the file cannot be found or cannot be parsed.

Example:

```
try {
    Properties prop = new Properties();
    prop.setProperty(
           Context.PROVIDER_URL, "ExampleConfigFile.xml");
    javax.naming.InitialContext context =
           new com.rti.naming.InitialContext(prop);
    // ...
} catch (NamingException nx) {
    // Respond to error...
}
```

Example:

```
try {
    Properties prop = new Properties();
    prop.setProperty(
           Context.PROVIDER_URL,
           "file:///opt/rtijms/example/ExampleConfigFile.xml");
    prop.setProperty(
           Context.INITIAL_CONTEXT_FACTORY,
           com.rti.jms.JmsConfigContextFactory.class.getName());
    javax.naming.InitialContext context =
           new javax.naming.InitialContext(prop);
    // ...
} catch (NamingException nx) {
    // Respond to error...
}
```

2.1.2.2 Looking Up Administered Objects

Method: **Object lookup(String name) throws javax.naming.NamingException**

Look up an administered object, whether a *ConnectionFactory* or a *Topic*, in the configuration file with this method. Names must include the name of the library followed by the name of the administered object; these two names can be delimited either with a forward slash ("/") or a double colon ("::").

If the same name is looked up multiple times, this context will return the same object every time, provided that the object is not garbage collected in between calls.

This method will throw a *NamingException* if an object of the given name is not present in the file or if the object cannot be instantiated.

Example:

In the file **ExampleConfigFile.xml**:

```
<?xml version="1.0" encoding="UTF-8"?>
<jms>
    <library name="Example">
            <connection_factory name="Example Connection Factory">
                    <!-- ... -->
            </connection_factory>
            <topic name="Example Topic">
                    <!-- ... -->
            </topic>
    </library>
</jms>
```

In source code:

```
try {
    ConnectionFactory factory =
            (ConnectionFactory) context.lookup(
                    "Example/Example Connection Factory");
    Topic topic =
            (Topic) context.lookup(
                    "Example::Example Topic");
    // ...
} catch (NamingException nx) {
    // Respond to error...
}
```

2.1.3 Programmatic Configuration

QoS parameters specified in a configuration file can be overridden programmatically by specifying their paths in the *Map* object passed to the *InitialContext* constructor. These paths take the form **com.rti.jms.qos:*<LibraryName>*::*<ObjectName>*::*<PathToField>***; a forward slash ("/") may also be used to delimit the path segments in place of the double colon. The property name prefix is identified by the constant **com.rti.naming.RTIContext.QOS_FIELD_PREFIX**.

Advanced: Two advanced features make it possible to make relatively complex QoS configuration changes at runtime:

- *The QoS value string can be an XML fragment*. If the string begins with '<', ends with '>', and is well-formed when wrapped in a root element (of arbitrary name), it will be interpreted as XML. If it is not surrounded in angle brackets, or if it cannot be parsed, the value will be treated as a literal string.
- *New XML elements can be added*, even when one or more elements with the same name already exist. To specify that a particular value should be added to the configuration, even if it already exists, prefix its name with '+'. This feature is important when working with elements in a list.

See below for examples of these features in action:

Example: Modifying an existing configuration

In the file **ExampleConfigFile.xml**:

```
<?xml version="1.0" encoding="UTF-8"?>
<jms>
    <library name="Example">
            <topic name="Example Topic">
                    <deadline>
                            <period>
                                    <sec>1</sec>
                                    <nanosec>0</nanosec>
                            </period>
                    </deadline>
            </topic>
    </library>
</jms>
```

In source code:

```
Properties prop = new Properties();
prop.setProperty(
    Context.PROVIDER_URL, "ExampleConfigFile.xml");
prop.setProperty(
    RTIContext.QOS_FIELD_PREFIX +
    ":Example::Example Topic::deadline::period::sec",
    "5");
// or:
// prop.setProperty(
//         RTIContext.QOS_FIELD_PREFIX +
//         ":Example/Example Topic/deadline/period/sec",
//         "5");
InitialContext context = new InitialContext(prop);
```

Example: Configuring a QoS that was not previously specified

In the file **ExampleConfigFile.xml**:

```
<?xml version="1.0" encoding="UTF-8"?>
<jms>
    <library name="Example">
           <topic name="Example Topic">
           </topic>
    </library>
</jms>
```

In source code:

```
Properties prop = new Properties();
prop.setProperty(
   Context.PROVIDER_URL, "ExampleConfigFile.xml");
prop.setProperty(
   RTIContext.QOS_FIELD_PREFIX +
   ":Example/Example Topic/deadline/period/sec",
   "5");
InitialContext context = new InitialContext(prop);
```

Example: Adding an XML fragment

In the file **ExampleConfigFile.xml**:

```
<?xml version="1.0" encoding="UTF-8"?>
<jms>
   <library name="Example">
    <connection_factory name="Example Factory">
      <property>
          <value>
            <element>
               <name>
                  dds.transport.UDPv4.builtin.parent.message_size_max
               </name>
               <value>65536</value>
            </element>
            <element>
               <name>
                  dds.transport.UDPv4.builtin.send_socket_buffer_size
               </name>
               <value>524288</value>
            </element>
          </value>
      </property>
    </connection_factory>
   </library>
</jms>
```

In source code:

```
Properties prop = new Properties();
prop.setProperty(
   Context.PROVIDER_URL, "ExampleConfigFile.xml");
prop.setProperty(
   RTIContext.QOS_FIELD_PREFIX +
    ":Example/Example Factory/property/value/+element",
    "<name>dds.transport.UDPv4.” +
    “builtin.recv_socket_buffer_size</name>" +
    "<value>2097152</value>");
InitialContext context = new InitialContext(prop);
```

2.2 Beginning Communication

In order to communicate with other applications over *RTI Message Service*, your application must first create a *Connection* to the network using a *ConnectionFactory* loaded from an *InitialContext* (see Loading a Configuration File (Section 2.1.2) for more information about the *InitialContext* class).

2.2.1 Connection Factory

Interface: javax.jms.ConnectionFactory

Interface: javax.jms.TopicConnectionFactory extends javax.jms.ConnectionFactory

A *ConnectionFactory* is a relatively lightweight object that represents a *Connection* configuration, which it instantiates in the form of one or more connections.

Your application does not instantiate connection factories directly. Connection factories are administered objects; they are looked up from an *InitialContext*. See Looking Up Administered Objects (Section 2.1.2.2).

All connection factories in *RTI Message Service* implement the *TopicConnectionFactory* interface in addition to the *ConnectionFactory* interface. See http://java.sun.com/javaee/5/docs/api/javax/jms/ConnectionFactory.html and http://java.sun.com/javaee/5/docs/api/javax/jms/TopicConnectionFactory.html for more information about these interfaces and the methods they define.

2.2.1.1 Creating a Connection

Method: javax.jms.Connection createConnection() throws javax.jms.JMSException

Method: javax.jms.TopicConnection createTopicConnection() throws javax.jms.JMSException

Method: javax.jms.Connection createConnection(String userName, String password) throws javax.jms.JMSException

Method: javax.jms.TopicConnection createTopicConnection(String userName, String password) throws javax.jms.JMSException

A *ConnectionFactory* instantiates new connections, all having the same QoS, using these factory methods. See Connection (Section 2.2.2).

All connections in *RTI Message Service* implement the *TopicConnection* interface in addition to the *Connection* interface, so the **createConnection** and **createTopicConnection** variants of these methods are equivalent.

Applications do not require a user name or password to join an *RTI Message Service* network, so the **userName** and **password** arguments, if present, are ignored.

2.2.2 Connection

Interface: javax.jms.Connection

Interface: javax.jms.TopicConnection extends javax.jms.Connection

A *Connection* is a heavyweight object representing an application's participation in an *RTI Message Service* network. In addition to resources allocated within the Java virtual machine (JVM), it also allocates native resources—including threads, sockets, and mutexes—outside of it.

Note: Although the number of connections an application may open is not artificially limited, developers are advised to create the minimum number necessary in order to maximize performance and minimize the CPU and network load on the system.

All connections in *RTI Message Service* implement the *TopicConnection* interface in addition to the *Connection* interface. See http://java.sun.com/javaee/5/docs/api/javax/jms/Connection.html and http://java.sun.com/javaee/5/docs/api/javax/jms/TopicConnection.html for more information about these interfaces and the methods they define.

2.2.2.1 Creating a Connection

Connections do not have public constructors; your application creates them using factory methods on a *ConnectionFactory*.

2.2.2.2 Creating a Session

Method: javax.jms.Session createSession(boolean transacted, int acknowledgeMode) throws javax.jms.JMSException

Method: javax.jms.Session createTopicSession(boolean transacted, int acknowledgeMode) throws javax.jms.JMSException

RTI Message Service does not support transacted sessions; **transacted** must be false. A **JMSException** will be thrown if it is not.

See Session (Section 2.2.3) for more information.

RTI Message Service supports the **AUTO_ACKNOWLEDGE** and **DUPS_OK_ACKNOWLEDGE** acknowledgement modes. For more information about these modes and how to avoid duplicate messages, see Chapter 6: Scalable High-Performance Applications: Message Reliability.

Example:

```
Connection myConnection = …;
try {
    Session mySession = myConnection.createSession(
            false, Session.DUPS_OK_ACKNOWLEDGE);
    // ...
} catch (JMSException jx) {
    // Handle exception
}
```

2.2.2.3 Starting and Stopping the Connection

Method: void start() throws javax.jms.JMSException

Method: void stop() throws JMSException

When a *Connection* is created, it is in the "stopped" state. When the *Connection* is stopped, no messages will be received. However, messages may be sent regardless of whether the *Connection* is started or stopped, and this state has no effect on the discovery status (see Chapter 2: Introduction to Peer-to-Peer Discovery).

2.2.2.4 Closing the Connection

Method: void close() throws javax.jms.JMSException

Closing a connection permanently halts communication, both the sending and receiving of messages, and indirectly closes any objects that may have been created, directly or indirectly, by that connection. These include sessions, message producers, and message consumers.

Some native resources are released by the execution of this method; others are not released until this connection is garbage collected. In fact, calling this method is optional in general; a connection will be closed when it is garbage collected. Because waiting for garbage collection decreases determinism, however, explicitly closing your connections is recommended.

2.2.3 Session

Interface: javax.jms.Session

Interface: javax.jms.TopicSession extends javax.jms.Session

Interface: com.rti.jms.RTISession extends javax.jms.TopicSession

A *Session* is a single-threaded context for sending and receiving messages. It serves as a factory for the objects that perform these tasks, which are of types *MessageProducer* and *MessageConsumer*, respectively.

All sessions in *RTI Message Service* implement the *TopicSession* interface in addition to the *Session* interface, so these two method variants are equivalent. See http://java.sun.com/javaee/5/docs/api/javax/jms/Session.html and http://java.sun.com/javaee/5/docs/api/javax/jms/TopicSession.html for more information about these interfaces and the methods they define.

Most *Session* methods pertain to either publishing or subscribing to messages. These methods are documented in Chapter 4: Publishing Messages and Chapter 5: Subscribing to Messages, respectively.

2.2.3.1 Creating a Session

Sessions do not have public constructors; your application creates them using factory methods on a *Connection*. See Section 2.2.3.

2.2.3.2 Closing a Session

Method: void close() throws javax.jms.JMSException

Closing a session implicitly closes all of the message producers and consumers of that session, ceasing application-level communication.

As for the resources of the session itself, some are released when the session is closed, while the tear-down of others waits until the object is garbage collected. In fact, calling this method is optional in general; a session will be closed when it is garbage collected. Because waiting for garbage collection decreases determinism, you should explicitly close your sessions (or their connection).

2.2.4 Examples: Putting it All Together

More extensive examples—incorporating file-based and programmatic configuration, connection factories, and connections—can be found in the **example** directory of your installation.

2.3 Responding to Network Events

Most production applications don't simply publish and subscribe blindly. Instead, they adapt their behavior to changes that occur on the network.

- ❏ Message producers and consumers may join or leave the network dynamically at any time. These events may represent opportunities to carry out certain initialization or policy enforcement logic. If an application leaves the network unexpectedly, it may represent an error condition to which your system will have to respond.
- ❏ New objects joining the network may be configured such that they can communicate with earlier-joining applications, or there may be configuration discrepancies that prevent that communication.
- ❏ Publishers and subscribers declare that they will hold to certain QoS contracts. For example, a publisher may promise to write data at a certain rate. If an object fails to hold to its declared contract, your application may need to respond.

This section provides an introduction to the status notification system provided by *RTI Message Service*.

2.3.1 Introduction to Status Notification

Interface: **javax.management.NotificationBroadcaster**

Interface: **javax.management.NotificationEmitter extends javax.management.NotificationBroadcaster**

Class: **com.rti.management.StatusNotifier implements javax.management.NotificationEmitter**

Class: **com.rti.management.Status extends javax.management.Notification**

The *StatusNotifier* class provides extensive information about distributed objects, either synchronously through a polled mechanism or asynchronously via a listener callback. It implements the JMX interface *NotificationEmitter*; that interface is documented further at http://java.sun.com/j2se/1.5.0/docs/api/javax/management/NotificationEmitter.html.

All notifications from a *StatusNotifier*, whether provided synchronously or asynchronously, are of the concrete type *Status*, which extends the JMX *Notification* class. This superclass is documented further at http://java.sun.com/j2se/1.5.0/docs/api/javax/management/Notification.html.

2.3.1.1 Creating a StatusNotifier

Constructor: StatusNotifier(javax.jms.Session session) throws javax.jms.JMSException

A *StatusNotifier* is associated with a *Session* and emits notifications pertaining to all of the publishers and subscribers operating within that session.

2.3.1.2 Deleting a StatusNotifier

Method: void com.rti.management.StatusNotifier.delete() throws javax.jms.JMSException

When your application no longer needs to obtain status information from the objects in a *Session*, it can delete that *StatusNotifier*.

2.3.2 Receiving Status Notifications

A *Status* object is distinguished by two pieces of information:

- Its *type*, a string that identifies the kind of status presented. The attributes that a *Status* has differs based on its type. For example, a *Status* describing incompatible QoS lists the number of incompatible objects found; this count is irrelevant to *Status* objects describing missed publication deadlines.
- Its *topic*, the destination to which the status pertains. For example, if the status indicates that a publisher has failed to meet its declared publication deadline, the status identifies the topic on which the publication failure occurred.

This section does not describe the individual notification types supported by the *StatusNotifier* class. Those notifications are described alongside the feature to which they pertain.

2.3.2.1 Polling for Status

Method: com.rti.management.Status getStatus(javax.jms.Topic topic, String notifType) throws javax.jms.JMSException

Method: com.rti.management.Status getStatus(javax.jms.Destination topic, String notifType) throws javax.jms.JMSException

These non-blocking methods return the session's current status of the given type as it pertains to the given topic. The two methods are functionally the same, since all destinations in *RTI Message Service* are of type *Topic*.

The PUBLICATION_MATCHED status used in the following example is described in detail in Chapter 4: Publishing Messages. The discovery process to which the example pertains is described in Introduction to Peer-to-Peer Discovery (Section 2.4) .

Example:

```
javax.jms.Session myProducerSession = …;
javax.jms.MessageProducer myProducer = …;
int numExpectedSubscribers = …;

StatusNotifier myNotifier = new StatusNotifier(myProducerSession)
Status pubMatch = myNotifier.getStatus(
    myProducer.getDestination(),
    StatusNotifier.PUBLICATION_MATCHED_NOTIFICATION_TYPE);
int numDiscoveredSubscribers = pubMatch.getIntAttribute(
    "currentCount");
if (numDiscoveredSubscribers >= numExpectedSubscribers) {
    // Discovery completed: do something...
} else {
    // Discovery not completed: do something else...
}
```

2.3.2.2 Receiving Status Notifications

Interface: **javax.management.NotificationListener**

Method: **void javax.management.NotificationListener.handleNotification(javax.management.Notification notification, Object handback)**

Method: **void javax.management.NotificationBroadcaster.addNotificationListener(javax.management.NotificationListener listener, javax.management.NotificationFilter filter, Object handback) throws IllegalArgumentException**

Method: **void javax.management.NotificationBroadcaster.removeNotificationListener(javax.management.NotificationListener listener) throws javax.management.ListenerNotFoundException**

Method: **void javax.management.NotificationEmitter.removeNotificationListener(javax.management.NotificationListener listener, javax.management.NotificationFilter filter, Object handback) throws javax.management.ListenerNotFoundException**

If your application needs to learn about status changes as soon as they occur, polling is not an appropriate approach. You need asynchronous notification of status changes. You register for these notifications using a *NotificationListener* attached to the *StatusNotifier*.

For minimum latency and maximum determinism, *RTI Message Service* dispatches status callbacks in internal middleware threads. For this reason, it is critical that applications not carry out expensive operations, such as blocking calls or long-running I/O, in the context of these callbacks. Applications that fail to adhere to this restriction can interfere with the correct operation of the middleware.

Example:

```
class MyListener implements NotificationListener {
    private int numExpectedSubscribers = …;

    public void handleNotification(
                   Notification notification, Object handback) {
           if (notification.getType.equals(
              StatusNotifier.PUBLICATION_MATCHED_NOTIFICATION_TYPE))
           {
                  Status status = (Status) notification;
                  int numDiscoveredSubscribers =
                                pubMatch.getIntAttribute(
                                              "currentCount");
                  if (numDiscoveredSubscribers >=
                         numExpectedSubscribers) {
                         // Discovery completed: do something...
                  } else {
                         // Discovery not completed:
                         // do something else...
                  }
           }
    }
}

javax.jms.Session myProducerSession = …;

StatusNotifier myNotifier = new StatusNotifier(myProducerSession)
NotificationListener myListener = new MyListener();
myNotifier.addNotificationListener(myListener, null, null);
// ...
myNotifier.removeNotificationListener(myListener);
```

2.3.3 Introspecting a Status

Each *Status* object contains a significant amount of information about the event that occurred, the object that triggered that notification, and the topic relative to which the

event occurred. Some of this information is common to all statuses; other attributes depend on the status's *type*.

2.3.3.1 Common Attributes

All *Status* objects provide the following information. This list is not exhaustive with respect to the methods inherited from **javax.management.Notification**; see http://java.sun.com/j2se/1.5.0/docs/api/javax/management/Notification.html for more information.

String Notification. getType()	A string identifying the event that occurred. Some status attributes are present or not depending on the value of this string; see below. The possible values of this string are constants defined by the *StatusNotifier* class.
String Notification. getMessage()	A human-readable description of the status.
Object Notification. getSource() **StatusNotifier Status.getSource()**	A reference to the *StatusNotifier* that emitted the status. The *Status* class overrides this method to provide stronger typing.
Topic Status.getTopic()	The topic to which the status pertains.

2.3.3.2 Type-Specific Attributes

Some *Status* attributes exist or not depending on the type string of that status. They are retrieved dynamically based on a string name.

Method: boolean getBooleanAttribute(String name)

Method: byte getByteAttribute(String name)

Method: char getCharAttribute(String name)

Method: double getDoubleAttribute(String name)

Method: float getFloatAttribute(String name)

Method: int getIntAttribute(String name)

Method: long getLongAttribute(String name)

Method: short getShortAttribute(String name)

Method: String getStringAttribute(String name)

Method: Object getAttribute(String name)

The **get<Type>Attribute** methods provide strongly typed attribute values for the case where the application knows that type ahead of time. They will throw a runtime exception if there is no valid (*i.e.* automatic widening) conversion from the attribute's type to the method's return type.

The generic **getAttribute** method is a convenience for use when the application does not know the concrete type of the attribute or wants to treat several types in a parallel way.

If an attribute of the given name does not exist, the **getAttribute** and **getStringAttribute** methods will return null. The **get<Primitive>Attribute** methods throw a runtime exception if the attribute doesn't exist.

2.4 Introduction to Peer-to-Peer Discovery

Applications that use *RTI Message Service* discover one another in an automatic, dynamic, peer-to-peer fashion; they do not require any centralized or per-node brokers in order to send messages.

2.4.1 Discovery and Matching

As part of this discovery process, applications automatically send announcements to one another when the following events occur:

- ❏ When a new *Connection* is created
- ❏ When a new *MessageProducer* or *MessageConsumer* is created
- ❏ When a *MessageProducer* or *MessageConsumer* is closed, either directly or indirectly because of a **Connection.close()** or **Session.close()** operation

When an application receives a notification that a producer or consumer has been created, it goes through a process called *matching*, in which the new producer or consumer is compared against the local consumers or producers to determine whether or not they can communicate. A producer and consumer are considered to match if:

- ❏ They are on the same *Topic* (see Chapter 3: Messages and Topics)
- ❏ They have compatible QoS (see the Configuration and Operation Manual)

Once a producer and consumer have been matched, messages published by the producer will begin to be received by the consumer. The time required for connections to discover one another and for producer-to-consumer matching to complete is on the order of one or two seconds for a modestly sized system, but times can vary greatly

based on the system size, the loads on the network and the target CPUs, and especially on how the discovery process is tuned.

When a producer or consumer is closed, its matches will be torn down, both locally and remotely, such that no further traffic will be sent to it. This "unmatching" is very fast, because the data channels over which the close is communicated already exist, unlike during the initial discovery, when more setup is required.

For more information about the discovery process, including how to tune it, consult the Configuration and Operation Manual.

2.4.2 Discovery-Related Notifications

An application can monitor and respond to changes in the discovery process as it proceeds.

Table 2.1 **Notification Type: StatusNotifier.PUBLICATION_MATCHED_NOTIFICATION_TYPE**

A publisher in this session has been matched—or unmatched—with a compatible subscriber.		
Attribute Name	**Attribute Type**	**Description**
totalCount	int	The total number of times that the publisher has discovered a matching subscriber since it was created.
totalCountChange	int	The change to the **totalCount** attribute since the last time this status was queried. If your application receives status notifications via a listener callback, this number will generally be 1. If your application polls for status changes, it may be take any integer value.
currentCount	int	The current number of subscribers with which the publisher is matched.
currentCountChange	int	The change to the **currentCount** attribute since the last time this status was queried. If your application receives status notifications via a listener callback, this number will generally be 1. If your application polls for status changes, it may be take any integer value.
currentCountPeak	int	The maximum number of subscribers to which the publisher has published simultaneously.

Table 2.2 **Notification Type: StatusNotifier.SUBSCRIPTION_MATCHED_NOTIFICATION_TYPE**

A subscriber in this session has been matched—or unmatched—with a compatible publisher.		
Attribute Name	**Attribute Type**	**Description**
totalCount	int	The total number of times that the subscriber has discovered a matching publisher since it was created.
totalCountChange	int	The change to the **totalCount** attribute since the last time this status was queried. If your application receives status notifications via a listener callback, this number will generally be 1. If your application polls for status changes, it may be take any integer value.
currentCount	int	The current number of publishers with which the subscriber is matched.
currentCountChange	int	The change to the **currentCount** attribute since the last time this status was queried. If your application receives status notifications via a listener callback, this number will generally be 1. If your application polls for status changes, it may be take any integer value.
currentCountPeak	int	The maximum number of publishers from which the subscriber has simultaneously received messages.

2.4.3 Example: Ensuring Delivery

In many applications, producers want to be sure that consumers are ready to receive messages before any messages are sent. Developers unfamiliar with RTI's dynamic discovery mechanism may be unsure of how they can avoid losing messages sent before matching has completed. This section describes several approaches.

2.4.3.1 Don't Worry, Decouple

In many cases, it doesn't actually matter to a message producer which—if any—consumers are available to receive a message. If producers and consumers are highly decoupled, starting and stopping at different times and with little bidirectional coordination across different data streams, just publish. If there are consumers ready to receive those messages, they will be received. The middleware knows whether the producer

has any matched consumers and will not put any packets on the network if there's no one to receive them.

Many one-way data feeds and sensor applications fall into this category.

2.4.3.2 Save It for Later

You may need to ensure that subscribers receive previously-published messages—all of them, perhaps, or the last *n*, or all of those published within a certain moving window of time—but you may want to decouple your publishing application from the knowledge of when subscribing applications start. In that case, you can configure the publisher to maintain published messages based on certain time and/or space limits; the middleware will provide those messages to late-joining subscribers automatically with no application intervention required.

This scenario is described in detail later in this manual in the chapters Message Reliability and Durability and Persistence for High Availability.

This communication model is widely applicable and may be the most familiar to users of brokered messaging solutions, in which centralized servers or per-node brokers are responsible for caching messages on behalf of individual subscribing applications.

2.4.3.3 Wait for Discovery

The durable data model described in Save It for Later (Section 2.4.3.2) may not be appropriate for applications that rely on highly deterministic latencies between the publishing of a message and the subscriber-side notification of that message's arrival. This is because historical messages will arrive in very quick succession, while the reception of live messages will be spaced in time according to the spacing of when they were sent. This behavior is illustrated in the following timeline:

Time	T0	T1	T2	T3	T4	T5	T6	T7	T8
Publisher	Start	Send M1	Send M2	Send M3	Send M4	Send M5	Send M6	Send M7	Send M8
Subscriber				Start	Receive M1, M2, M3	Receive M4	Receive M5	Receive M6	Receive M7

If this behavior is undesirable, publishing applications can wait for subscribing applications to start before sending any messages.

Example:

```
javax.jms.Session myProducerSession = …;
javax.jms.MessageProducer myProducer = …;
int numExpectedSubscribers = …;
Message myMessage = …;

com.rti.management.StatusNotifier myNotifier =
    new StatusNotifier(myProducerSession)

while (true) {
    com.rti.management.Status pubMatch = myNotifier.getStatus(
           myProducer.getDestination(),
           com.rti.management.StatusNotifier.
                  PUBLICATION_MATCHED_NOTIFICATION_TYPE);
    int numDiscoveredSubscribers = pubMatch.getIntAttribute(
           "currentCount");
    if (numDiscoveredSubscribers >= numExpectedSubscribers) {
           break;
    }
    Thread.sleep(500); // half second
}
myProducer.send(myMessage);
```

In the example above, the application polls for discovery status changes. It is also possible to register for asynchronous notifications. For more information about the status notification capabilities of *RTI Message Service*, see Responding to Network Events (Section 2.3).

The approach demonstrated in the example code may work by itself in many cases. However, there is a subtle race condition that may reveal itself if the subscriber is much more heavily loaded than the publisher: the publisher may finish matching with the subscriber, but the subscriber may not have finished matching with the publisher. In such a case, messages will be discarded on the subscriber side without being delivered until the subscriber has completed the matching process. There are two ways to definitively avoid this situation:

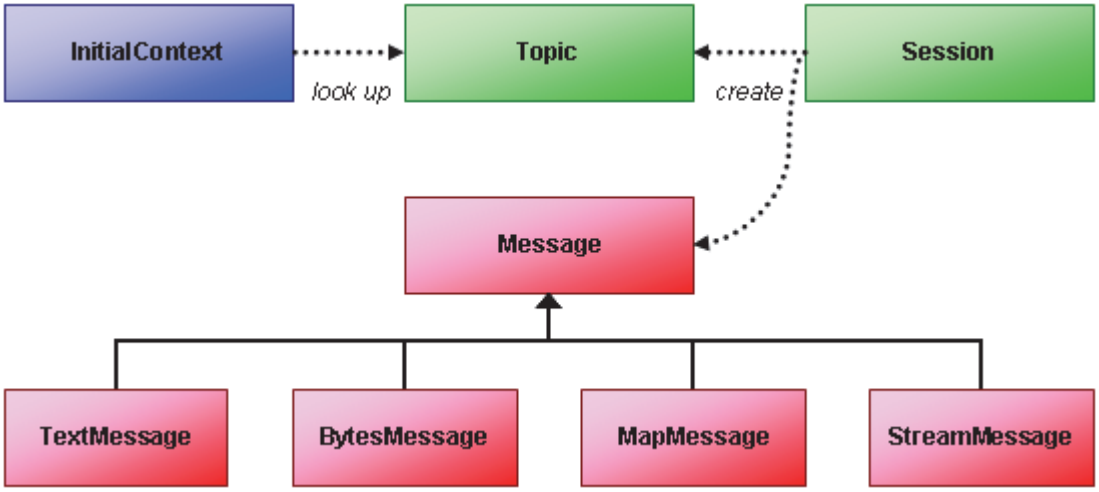
- Combine this approach with that described in Save It for Later (Section 2.4.3.2). You may still observe the trend described in the timeline above, but it should be much less pronounced.

- Use a second topic, from subscriber to publisher, to indicate that the subscriber is ready to receive data on the primary topic. This control topic should be durable, as described in Save It for Later (Section 2.4.3.2).

.

Chapter 3 Messages and Topics

RTI Message Service is a publish-subscribe messaging middleware. This chapter describes in detail the publish-subscribe communication paradigm and the various message types provided by the API.

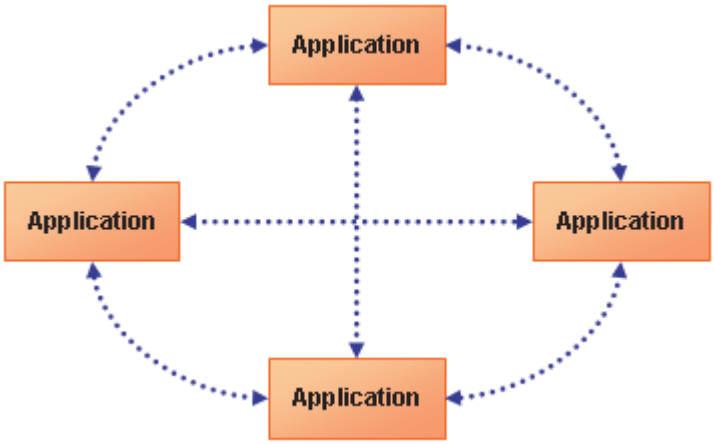


This chapter is organized as follows:

3.1 Introduction to Publish-Subscribe Communication

In the publish-subscribe communications model, applications "subscribe" to data they need and "publish" data they want to share. Publish-subscribe communication architectures are ideal for distributing large quantities of time-sensitive information efficiently to multiple recipients. Examples of publish-subscribe systems in everyday life include television, magazines, and newspapers.

In point-to-point and client-server network topologies, data producers and consumers are closely coupled, and applications must typically explicitly manage the connections between them.



In a publish-subscribe system, on the other hand, applications treat the network as a shared bus. The middleware takes responsibility for multiplexing data from each publisher to an arbitrary number of subscribers. This layer of abstraction allows developers and integrators to introduce transport-related optimizations—such as using multicast addressing instead of unicast or introducing higher-performing link technology—without changing core application logic.

In most publish-subscribe messaging systems, messages are brokered by centralized and/or per-node servers on their way from publisher to subscriber. In *RTI Message* Service, applications communicate peer-to-peer; messages pass directly between the publisher and the subscribers.

RTI Message Service supports other mechanisms that go beyond the basic publish-subscribe model. One key benefit is that applications that use *RTI Message Service* for their communications are entirely decoupled. Very little of their design time has to be spent on how to handle their mutual interactions. In particular, applications never need information about the other participating applications, including their existence or locations. *RTI Message Service* automatically handles all aspects of message delivery, without requiring any intervention from the user applications, including:

- ❏ determining who should receive the messages,
- ❏ where recipients are located, and
- ❏ what happens if messages cannot be delivered.

This is made possible by how *RTI Message Service* allows the user to specify Quality of Service (QoS) parameters as a way to configure automatic discovery mechanisms and specify the behavior used when sending and receiving messages. By exchanging messages in a completely anonymous manner, *RTI Message Service* facilitates system integration and flexible deployment.

3.2 Working with Messages

Most messaging middleware APIs, including the JMS API implemented by *RTI Message Service*, use pre-defined "message" objects. JMS defines several types of messages, each optimized for carrying a certain type of payload.

3.2.1 Sending and Receiving Messages without Bodies

Interface: javax.jms.Message

The *Message* interface is the base of the message type hierarchy. All messages have certain well-known header fields, an arbitrary collection of key-value properties, and an optional body that varies according to the message's concrete base type.

See http://java.sun.com/javaee/5/docs/api/javax/jms/Message.html for more information about the *Message* interface and the methods it defines.

Method: javax.jms.Message javax.jms.Session.createMessage() throws javax.jms.JMSException

Applications use a factory method on a *Session* to create a new *Message* without a body. Such messages may be useful for simple notifications. More complex data will likely require a more capable Message sub-type; see the following sections.

3.2.2 Sending and Receiving Text Strings

Interface: javax.jms.TextMessage extends javax.jms.Message

The body of a *TextMessage* is a single string. This is the simplest of the *Message* subtypes.

See http://java.sun.com/javaee/5/docs/api/javax/jms/TextMessage.html for more information about the *TextMessage* interface and the methods it defines.

Method: javax.jms.TextMessage javax.jms.Session.createTextMessage() throws javax.jms.JMSException

Method: javax.jms.TextMessage javax.jms.Session.createTextMessage(String text) throws javax.jms.JMSException

Applications use factory methods on a *Session* to create new text messages.

3.2.3 Sending and Receiving Opaque Byte Buffers

Interface: javax.jms.BytesMessage extends javax.jms.Message

The body of a *BytesMessage* contains a variable-length stream of bytes. The interface provides **read** and **write** methods that encode and decode primitive types, strings, and byte arrays to and from this stream.

BytesMessage provides the most-efficient, lowest-overhead mechanism for transporting data from one application to another. However, this efficiency comes at a cost: once data has been encoded into a *BytesMessage*, it no longer contains any type information, so it cannot be introspected. Recipients must have *a priori* knowledge of a message's structure.

See http://java.sun.com/javaee/5/docs/api/javax/jms/BytesMessage.html for more information about the *BytesMessage* interface and the methods it defines.

Method: javax.jms.BytesMessage javax.jms.Session.createBytesMessage() throws javax.jms.JMSException

Applications use a factory method on a *Session* to create a new *BytesMessage*.

3.2.4 Sending and Receiving Key-Value Pairs

Interface: javax.jms.MapMessage extends javax.jms.Message

The body of a *MapMessage* contains an arbitrary set of strongly typed key-value pairs.

MapMessage provides an easy-to-use representation for complex data structures and allows these structures to be introspected. However, the extensive type information that accompanies the application data makes the on-the-network representation of map messages larger than that of messages of other types.

See http://java.sun.com/javaee/5/docs/api/javax/jms/MapMessage.html for more information about the *MapMessage* interface and the methods it defines.

Method: javax.jms.MapMessage javax.jms.Session.createMapMessage() throws javax.jms.JMSException

Applications use a factory method on a *Session* to create a new *MapMessage*.

3.2.5 Sending and Receiving Streams of Typed Values

Interface: javax.jms.StreamMessage extends javax.jms.Message

The body of a *StreamMessage* contains a variable-length sequence of strongly typed primitive, string, and byte array values.

See http://java.sun.com/javaee/5/docs/api/javax/jms/StreamMessage.html for more information about the *StreamMessage* interface and the methods it defines.

Method: javax.jms.StreamMessage javax.jms.Session.createStreamMessage() throws javax.jms.JMSException

Applications use a factory method on a *Session* to create a new *StreamMessage*.

3.3 Working with Topics

In a distributed application built using point-to-point network connections, data is sent to—and received from—physical network addresses. Such connections are brittle, because as application components and subsystems move from node to node, or as fan-out changes occur (that is, as a producer is associated with more or fewer consumers), the application code and/or configuration needs to change.

A topic, in contrast, is a named virtual destination to which messages can be sent and from which those messages can be received. The relationships between logical destinations and the underlying network addressing system are maintained by the middleware, isolating application code from this concern.

3.3.1 Topics

Interface: javax.jms.Topic extends javax.jms.Destination

A *Topic* is a named *Destination* that is used to create publish-subscribe message producers and consumers. See http://java.sun.com/javaee/5/docs/api/javax/jms/Topic.html and http://java.sun.com/javaee/5/docs/api/javax/jms/Destination.html for more information about these interfaces and the methods they define.

3.3.1.1 Creating or Looking Up a Topic

Method: Object com.rti.jms.InitialContext.lookup(String objectName) throws javax.naming.NamingException

Method: javax.jms.Topic javax.jms.Session.createTopic(java.lang.String topicName) throws javax.jms.JMSException

Method: javax.jms.Topic javax.jms.TopicSession.createTopic(java.lang.String topicName) throws javax.jms.JMSException

There are two ways of instantiating a *Topic* in your application:

Looking up a previously-defined administrated *Topic* using an *InitialContext*. This method is described in Configuring the Middleware (Section 2.1).

Creating a new *Topic* programmatically using a factory method on a *Session*.

With respect to how a given *Topic* can be used within your application, these two capabilities are entirely equivalent. However, the QoS that govern communication between publishers and subscribers using the *Topic* could be different.

The QoS of an administered topic is defined in a configuration file, allowing system designers and administrators to customize that QoS as required for a given application. All users of such a topic will use these pre-defined QoS settings.

When creating a *Topic* programmatically, the middleware is responsible for determining the QoS for that topic, not an administrator. If it is able to locate a topic of the given name in the configuration file, it will use the QoS specified there. In such a case, **createTopic** and **lookup** will be equivalent. Otherwise, all QoS policies will take default values.

To ensure that QoS are deterministically determined at runtime and configuration-controlled across application launches, RTI recommends that most applications look up topics from an *InitialContext* rather than creating them from a *Session*.

3.3.1.2 Topic Names

Method: String getTopicName() throws javax.jms.JMSException

A topic's name is defined when that topic is specified in a configuration file (see Configuring the Middleware (Section 2.1)) or created programmatically (see above), and it can be retrieved later using this method. The name is a literal string, unique within an *RTI Message Service* network, and must be no more than 255 characters in length.

3.3.2 Temporary Topics

Interface: javax.jms.TemporaryTopic extends javax.jms.Topic

A temporary topic is a unique topic that exists only for the lifetime of a given *Connection* and can only be consumed by that *Connection*. Its name is chosen by the middleware. See http://java.sun.com/javaee/5/docs/api/javax/jms/TemporaryTopic.html for more information about this interface and the methods is defines.

A temporary topic always has default QoS values.

3.3.2.1 Creating a Temporary Topic

Method: javax.jms.TemporaryTopic javax.jms.Session.createTemporaryTopic() throws javax.jms.JMSException

Method: javax.jms.TemporaryTopic javax.jms.TopicSession.createTemporaryTopic() throws javax.jms.JMSException

Temporary topics are created using a factory method on a *Session*.

3.3.2.2 Deleting a Temporary Topic

Method: void delete() throws javax.jms.JMSException

Unlike other topics, temporary topics can be deleted—provided that there are no producers or consumers using it—indicating that the application no longer intends to use that temporary topic.

3.4 Advanced: Keyed Topics for Real-Time Performance and Scalability

A topic defines an extensive set of QoS that govern the communication between the publishers and subscribers that use that topic. The application- and system-level requirements that those QoS enforce are generally common to a large number of similar objects in the world, to which middleware messages pertain.

For example, a market data distribution system may disseminate information about many different stocks. However, most of the QoS pertaining to that stock data are the same, regardless of that the particular stock is: the level of reliability required, the amount of historical data that will be kept and on what basis, and so forth. It is very unlikely that the market data infrastructure will want to distribute Apple and Intel data reliably but Microsoft and IBM data in a best-effort fashion, for instance.

For example, a radar track management system may track very many objects. These objects have distinct identities, but the system will track them all in a consistent way. And unlike stocks, the set of objects tracked by a radar system is open; it is not possible to know ahead of time all of the objects that might be detected. New objects can appear at any time, and the same object can come and go quickly and repeatedly. Setting up and tearing down topics and the message producers and consumers that use them very rapidly is not an efficient use of CPU and network resources.

For these reasons, it is often best to consider a topic to be a relatively coarse-grained communication pathway. Using separate topics for fine-grained information—such as for each stock or each flight detected by a radar system—can lead to heavy resource usage and duplicated configuration data.

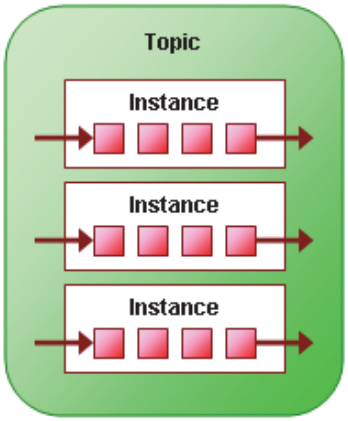
Although different real-world objects may be governed by the same set of requirements, it is desirable for the middleware to distinguish among them. Returning to the market data distribution system example, there may be a requirement to maintain the latest price at all times so that new subscribers can access it immediately. But this price should be maintained *separately for each stock*; a single last price, which could refer to Apple at one moment and IBM the next, would not be very useful.

Relational databases address this critical use case with a concept called a *key*: records pertaining to multiple objects of the same type can be stored in the same table; a field called the "key" distinguishes them from one another. *RTI Message Service* takes the same approach.

In *RTI Message Service*, the key is a well-known string property in a message. If no value has been set, the empty string is assumed.

```
myMessage.setStringProperty("JMS_RTIKey", "the key value");
```

The messages that share a key value are referred to as an *instance*.



The number of historical messages that the middleware maintains can be configured on a per-instance basis. This limit is referred to as the "history depth." If sufficient new messages of the same instance arrive to exceed the history depth before the subscriber has read previous messages, the oldest messages will be discarded[1]. This ability—to "cancel" the delivery of obsolete messages—is critical in highly deterministic real-time applications. It allows subscribers to spend more of their time processing valid data and less of it processing obsolete data, decreasing latency, increasing determinism, and decreasing the chance that the subscriber will fall behind the publisher, which can lead to problems of its own.

To enable per-instance message management, a topic must be marked as *keyed* in its configuration file:

```
<topic name="SampleTopic">
    <keyed>true</keyed>
    <!-- ... -->
</topic>
```

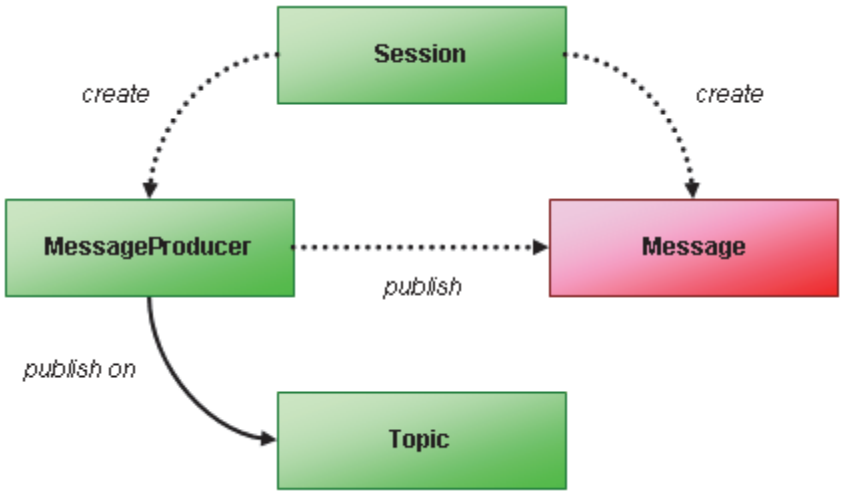
If the **<keyed></keyed>** property is not specified, it is assumed to have the value *false*.

For more information about this capability, see Chapter 8: Scalable High-Performance Applications: Keys.

1. For more information about RTI's history feature, see Chapter 6: Scalable High-Performance Applications: Message Reliability and Chapter 7: Scalable High-Performance Applications: Durability and Persistence for High Availability.

Chapter 4 Publishing Messages

This chapter guides you through all of the steps you need in order to start publishing messages, including the important objects and their methods. Before reading this chapter, you should have read the Getting Started Guide and gone through the tutorial in it. That tutorial showed you the code you need to publish data; the goal of this chapter is to help you understand that code.



This chapter is organized as follows:

- Step-by-Step Overview (Section 4.1)
- Message Producer (Section 4.2)

4.1 Step-by-Step Overview

In order to publish messages, you will need to go through these steps:

From Chapter 2: Connecting to the Network:

1. Create a configuration file and define a topic and connection factory in it.
2. Create an **InitialContext** and use it to look up the **ConnectionFactory** you defined.
3. Use the **ConnectionFactory** to create a **Connection**.
4. Use the **Connection** to create a **Session**.

The steps above are the same for publishing and subscribing applications.

From Chapter 3: Messages and Topics:

1. Look up the **Topic** you defined.
2. Create a **Message** object.

Described in this chapter:

3. Use the **Session** and the **Topic** to create a **MessageProducer**.
4. Use the **MessageProducer** to send the **Message**.

4.2 Message Producer

Interface: javax.jms.MessageProducer

Interface: javax.jms.TopicPublisher extends javax.jms.MessageProducer

Interface: com.rti.jms.RTIMessageProducer extends javax.jms.TopicPublisher

A **MessageProducer** publishes messages on a specified **Topic** according to a specified set of QoS policies.

All producers in *RTI Message Service* implement the **MessageProducer**, **TopicPublisher**, and **RTIMessageProducer** interfaces. The latter is described here; more information about the former, and the methods they define, can be found here:

- http://java.sun.com/javaee/5/docs/api/javax/jms/MessageProducer.html
- http://java.sun.com/javaee/5/docs/api/javax/jms/TopicPublisher.html

4.2.1 Creating a Message Producer

Method: **javax.jms.MessageProducer javax.jms.Session.createProducer(javax.jms.Destination destination) throws javax.jms.JMSException**

Method: **javax.jms.TopicPublisher javax.jms.TopicSession.createPublisher(javax.jms.Topic topic) throws javax.jms.JMSException**

Your application creates a message producer for each **Topic** it wishes to publish using factory methods on a **Session** object. The **Destination**/**Topic** passed to these methods cannot be null.

A **Session** may create any number of producers. It is even possible to create multiple producers for the same topic, although doing so is typically not useful.

4.2.2 Closing a Message Producer

Method: **void close() throws javax.jms.JMSException**

Closing a message producer causes it to release any resources it's using, including native resources. Once a producer has been closed, it can no longer be used to publish messages.

4.2.3 Publishing Messages

Method: **void javax.jms.MessageProducer.send(javax.jms.Message message) throws javax.jms.JMSException**

Method: **void javax.jms.TopicPublisher.publish(javax.jms.Message message) throws javax.jms.JMSException**

Method: **void javax.jms.MessageProducer.send(javax.jms.Message message, int deliveryMode, int priority, long timeToLive) throws javax.jms.JMSException**

Method: **void javax.jms.MessageProducer.send(javax.jms.Destination topic, javax.jms.Message message) throws javax.jms.JMSException**

Method: **void javax.jms.TopicPublisher.publish(javax.jms.Topic topic, javax.jms.Message message) throws javax.jms.JMSException**

Method: **void javax.jms.MessageProducer.send(javax.jms.Destination topic, javax.jms.Message message, int deliveryMode, int priority, long timeToLive) throws javax.jms.JMSException**

Method: **void javax.jms.TopicPublisher.publish(javax.jms.Topic topic, javax.jms.Message message, int deliveryMode, int priority, long timeToLive) throws javax.jms.JMSException**

Publish a message to all compatible subscribers on the specified topic.

- If no destination (topic) is specified, the message will be published to this publisher's own topic. This is the typical case.
- If a destination is specified, the message will be published to that topic. A publisher to that topic must have been previously created within the current session; this call to `send` or `publish` will behave as if the message had been published using that other publisher.

Parameter: deliveryMode

The delivery mode cannot be changed on a per-message basis. If specified, the delivery mode must match that of the publisher.

Parameter: priority

Your application may provide a priority value to accompany the message, as specified by the JMS specification. However, the streamlined architecture of *RTI Message Service* makes this priority irrelevant; it will be ignored.

- In most cases, this publisher will put the message on the network within the context of this call. Because there are no intermediate brokers needed to route this message to its consumers, there is no intermediate processing to prioritize.
- Likewise, on the subscribing side, the message will be made available to the application as soon as it is read from the socket and ordered relative to other received messages. If the receiving application is using a **MessageListener**, this delivery will occur within the context of the thread that is reading that socket; once again, there is no intermediate processing or queuing to prioritize.

Parameter: timeToLive

The default behavior of the publisher is to apply its own time-to-live QoS to all of the messages it publishes. While it is possible to override this value on a per-message basis, doing so is only a hint to the middleware; the middleware may not be able to honor it. For the most deterministic behavior, applications are advised to always use the publisher's own time-to-live configuration rather than overriding it.

4.2.4 Coherent Changes

Method: void com.rti.jms.RTISession.beginCoherentChanges() throws javax.jms.JMSException

Method: void com.rti.jms.RTISession.endCoherentChanges() throws javax.jms.JMSException

These methods, which are extensions to the JMS API, allow any **Session** to gather a set of outgoing messages—potentially across multiple publishers—into a lightweight transaction ("coherent set"), such that subscribing applications will receive either all of the messages or none of them.

These calls can be nested. In that case, the coherent set terminates only with the last call to **endCoherentChanges()**.

The support for coherent changes enables a publishing application to describe several state changes—that could be propagated on the same or different topics—and have those changes be seen atomically by subscribers. This is useful in cases where the values are inter-related. For example, if there are two messages representing the altitude and velocity of the same aircraft, it may be important to communicate those values such that the subscriber can see both together; otherwise, it may e.g., erroneously interpret that the aircraft is on a collision course.

Example:

```
((RTISession) mySession).beginCoherentChanges();
try {
    myPublisher1.send(myMessage1);
    myPublisher2.send(myMessage2);
    myPublisher3.send(myMessage3);
} finally {
    ((RTISession) mySession).endCoherentChanges();
}
```

4.2.5 Delivery Mode and Persistent Publication

Method: int getDeliveryMode() throws javax.jms.JMSException

Method: void setDeliveryMode(int deliveryMode) throws javax.jms.JMSException

Every message producer has a *delivery mode* that indicates whether the messages it sends will be saved to persistent storage in order to provide high availability if the producer should fail. In *RTI Message Service*, this mechanism is connected to a broader infrastructure for durability and persistence that includes both publishers and subscribers. If your application requires persistent messaging, it is highly recommended that you read about this topic in detail in Chapter 7: Scalable High-Performance Applications: Durability and Persistence for High Availability, which includes other closely related QoS. The description here is only a brief summary of this capability as it pertains to publisher-side delivery mode specifically.

The publisher supports the durability kinds listed in Table 4.1.

Table 4.1 **Durability Kinds**

<table>
<tr><th>RTI Durability Kind</th><th>Equivalent Delivery Mode</th><th>Configuration</th></tr>
<tr><td rowspan="2">VOLATILE</td><td>NON_PERSISTENT</td><td><pre><topic name="SampleTopic">
    <producer_defaults>
        <durability>
            <kind>
VOLATILE_DURABILITY_QOS
            </kind>
        </durability>
    </producer_defaults>
</topic></pre></td></tr>
<tr><td colspan="2">The middleware does not need to keep any message on behalf of any subscriber that is unknown to the publisher at the time the message is written. Once a message has been acknowledged by all known subscribers, it can be removed from the publisher's cache. If the publisher fails before the message is acknowledged, the message could be lost.
This is the highest performing configuration and the default.</td></tr>
</table>

Table 4.1 **Durability Kinds**

<table>
<tr><th>RTI Durability Kind</th><th>Equivalent Delivery Mode</th><th>Configuration</th></tr>
<tr><td rowspan="2">TRANSIENT_LOCAL</td><td>NON_PERSISTENT</td><td><pre><topic name="SampleTopic">
    <producer_defaults>
        <durability>
            <kind>
        TRANSIENT_LOCAL_DURABILITY_QOS
            </kind>
        </durability>
    </producer_defaults>
</topic></pre></td></tr>
<tr><td colspan="2">The publisher will keep some number of historical messages so that they can be delivered to any potential late-joining subscribers. However, the messages will not be persisted to permanent storage, so if the publisher fails before the message is acknowledged, the message could be lost.</td></tr>
<tr><td rowspan="2">TRANSIENT</td><td>NON_PERSISTENT</td><td><pre><topic name="SampleTopic">
   <producer_defaults>
      <durability>
         <kind>
            TRANSIENT_DURABILITY_QOS
         </kind>
      </durability>
   </producer_defaults>
</topic></pre></td></tr>
<tr><td colspan="2">The publisher will keep some number of historical messages so that they can be delivered to any potential late-joining subscribers. These messages will be retained in a persistence service external to the original publisher, so that if the publisher fails, the message will remain available. However, the message may not be persisted to permanent storage, so if the publisher and the service both fail, the message could be lost.</td></tr>
</table>

Table 4.1 **Durability Kinds**

RTI Durability Kind	Equivalent Delivery Mode	Configuration

PERSISTENT	PERSISTENT	`<topic name="SampleTopic">`
		`  <producer_defaults>`
		`  <durability>`
		`  <kind>`
		`  PERSISTENT_DURABILITY_QOS`
		`  </kind>`
		`  </durability>`
		`  </producer_defaults>`
		`</topic>`

	Messages are kept in permanent storage external to the original publisher so that they can outlive multiple failures or even a full system restart.	

These QoS policies are specified in the configuration file loaded by the application. Once the **InitialContext** has been created, they cannot be changed. The **setDeliveryMode** method will throw an exception if its argument does not match the current delivery mode.

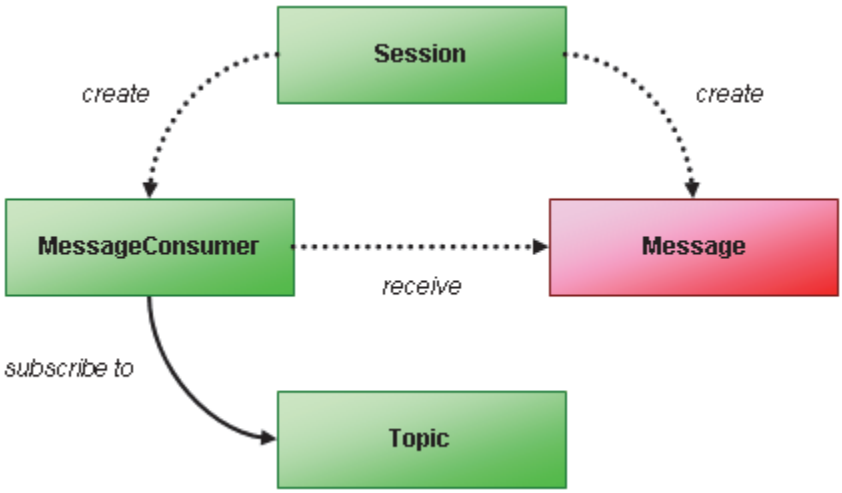
4.2.6 Batching Messages for Lower Overhead and Increased Throughput

When your application needs to send small messages (less than a couple of kilobytes) at a high rate, your operating system's network stack may become a bottleneck. *RTI Message Service* can erase this bottleneck by "batching" many messages into a single network packet, potentially increasing throughput several-fold. However, batching may not be appropriate for all scenarios; for example, if your application sends messages only sporadically, the batching process may introduce unacceptable latency.

For more information about this capability, including how to configure it, consult Chapter 5, "Throughput Management," in the Configuration and Operation Manual.

Chapter 5 Subscribing to Messages

This chapter explains how to start subscribing to messages, including the important objects and their methods. Before reading this chapter, you should have read the Getting Started Guide and gone through the tutorial in it. The tutorial shows you the code needed to subscribe to data; the goal of this chapter is to help you understand that code.



This chapter is organized as follows:

- Step-by-Step Overview (Section 5.1)
- Message Consumer (Section 5.2)
- Message Filtering (Section 5.3)

5.1 Step-by-Step Overview

In order to receive messages, you will need to go through these steps:

From Chapter 2: Connecting to the Network:

1. Create a configuration file and define a topic and connection factory in it.
2. Create an **InitialContext** and use it to look up the **ConnectionFactory** you defined.
3. Use the **ConnectionFactory** to create a **Connection**.
4. Use the **Connection** to create a **Session**.

The steps above are the same for publishing and subscribing applications.

From Chapter 3: Messages and Topics:

1. Look up the **Topic** you defined.

Described in this chapter:

2. Use the **Session** and the **Topic** to create a **MessageConsumer**.
3. Use the **MessageConsumer** to receive messages synchronously, or install a **MessageListener** to receive them asynchronously.

5.2 Message Consumer

Interface: javax.jms.MessageConsumer

Interface: javax.jms.TopicSubscriber extends javax.jms.MessageConsumer

A **MessageConsumer** subscribes to messages on a specified **Topic** according to a specified set of QoS policies.

All consumers in *RTI Message Service* implement the **MessageConsumer** and **TopicSubscriber** interfaces. More information about these interfaces and the methods they define can be found at

- http://java.sun.com/javaee/5/docs/api/javax/jms/MessageConsumer.html
- http://java.sun.com/javaee/5/docs/api/javax/jms/TopicSubscriber.html

5.2.1 Creating a Message Consumer

Method: javax.jms.MessageConsumer javax.jms.Session.createConsumer(javax.jms.Destination topic) throws javax.jms.JMSException

Method: javax.jms.MessageConsumer javax.jms.Session.createConsumer(javax.jms.Destination topic, String messageSelector) throws javax.jms.JMSException

Method: javax.jms.TopicSubscriber javax.jms.TopicSession.createSubscriber(javax.jms.Topic topic) throws javax.jms.JMSException

Your application creates a message consumer for each **Topic** to which it wishes to subscribe using factory methods on a **Session** object. The **Destination**/**Topic** passed to these methods cannot be null.

A **Session** may create any number of consumers. It is even possible to create multiple consumers for the same topic, although doing so is typically not useful.

Some factory method variants accept a content-based filter called a "message selector." This argument is discussed in Content-Based Filtering (Section 5.3.1).

5.2.2 Closing a Message Consumer

Method: void close() throws javax.jms.JMSException

Closing a message consumer causes it to release any resources it's using, including native resources. Once a consumer has been closed, it can no longer be used to receive messages.

5.2.3 Receiving Messages

There are two ways to receive messages in *RTI Message Service*: synchronously, using a variant of the receive method, or asynchronously, using a **MessageListener**. Which mechanism your application uses depends on its requirements and constraints.

5.2.3.1 Receiving Messages Synchronously

Method: javax.jms.Message receive() throws javax.jms.JMSException

Method: javax.jms.Message receive(long timeout) throws javax.jms.JMSException

Method: javax.jms.Message receiveNoWait() throws javax.jms.JMSException

The **receive** method blocks for a period of time, waiting for a message to arrive. When a message does arrive, this method will unblock and return a copy of it to the application. This **Message** object belongs to the caller and can be used in any way.

- ❏ The no-argument variant of this method will block indefinitely if no message arrives and the consumer is not closed.
- ❏ The variant that accepts a timeout will block only for a finite period of time; the timeout is measured in milliseconds. This method will return null if the timeout expires without the arrival of a message.
- ❏ The **receiveNoWait** method will return a **Message** immediately if one is already available, but it will never block.

Synchronous message reception is simpler than asynchronous reception: there are fewer calls to make, it does not require dealing directly with multiple threads, and there are fewer restrictions on how the resulting messages can be used.

However, synchronous reception may not be appropriate for those data streams with the most demanding performance requirements. It requires the allocation of a new **Message** object with each call, and a context switch interposes itself in between reading the message from the network and handing that message off to the application. Both of these steps add latency and sacrifice determinism.

Example:

```
MessageConsumer myConsumer = …;

try {
    Message myMessage = myConsumer.receive();
    // Handle message...
} catch (JMSException jx) {
    // Respond to exception...
}
```

5.2.3.2 Receiving Messages Asynchronously

Interface: **javax.jms.MessageListener**

Method: **javax.jms.MessageListener javax.jms.MessageConsumer.getMessageListener() throws javax.jms.JMSException**

Method: **void javax.jms.MessageConsumer.setMessageListener(javax.jms.MessageListener listener) throws javax.jms.JMSException**

Method: **void javax.jms.MessageListener.onMessage(javax.jms.Message message)**

A **MessageListener** delivers a **Message** immediately after it is read from the network and in the same thread. The **Message** that is provided in the **onMessage** callback is not a copy; it is owned by an internal middleware pool and may be reused. It is therefore critical that applications not use messages received in this way outside of the callback. Furthermore, because the callback is issued within a middleware thread, it is important that applications limit the amount of processing they perform in the callback. Issuing blocking calls or performing costly I/O tasks in particular could interfere with middleware operations.

Because this delivery mechanism requires no object allocation, copying, or context switch, it generally performs better than synchronous message reception. However, the restrictions it imposes may make it inappropriate for some use cases.

Example:

```
class MyMessageListener implements MessageListener {
    void onMessage(Message message) {
            // Handle message...
    }
}

MessageConsumer myConsumer = …;
MessageListener myListener = new MyMessageListener();
try {
    myConsumer.setMessageListener(myListener);
} catch (JMSException jx) {
    // Respond to exception...
}
```

5.3 Message Filtering

Your application may not be interested in all messages that are published on a topic.

- ❑ It may be interested only in certain application events, for example, when a message arrives with a certain correlation ID.
- ❑ It may need to take care that a faster publisher does not overload it with extraneous data.
- ❑ Your distributed system may incorporate a complex event processing (CEP) engine, and you want to offload some of the CPU burden on that component by distributing a first level of filtering across your system.

RTI Message Service provides extensive filtering capabilities to address these use cases.

5.3.1 Content-Based Filtering

When your application creates a message consumer, it can specify a *message selector*, a filter that will be applied to all messages on the consumer's topic. Only messages that pass the filter will be delivered to your application.

5.3.1.1 Filter Syntax

The syntax for the filter is a subset of SQL with some extensions. The filter is the part of a SQL statement that follows the **WHERE** keyword. Table 5.1 lists the supported operations.

Table 5.1 **Supported Filter Operations**

Operator	Description	Examples
'=', '<>'	Message field value is equal to (or not equal to) another value.	`JMSCorrelationID = 'foo'`
'>', '>=', '<', '<='	Message field value has the specified inequality relationship with another value.	`JMSCorrelationID < 'aabbcc'`

Table 5.1 **Supported Filter Operations**

'MATCH'	Message field value matches a dot-delimited set of regular expressions. This operator may only be used with string operands, where the right-hand operator is a constant string *pattern*. A string pattern specifies a template that the field value must match. MATCH is case-sensitive. The syntax is similar to the POSIX **fnmatch** syntax (1003.2-1992 section B.6), but with the role of the character '/' replaced by '.'. This type of filter may also be familiar to TIBCO Rendezvous and SmartSockets users.	
	The question mark ('?') character matches 0 or 1 characters in the field value.	`JMS_RTIKey MATCH 'ab?c'`
	The asterisk ('*') character matches 0 or more characters in the field value.	`JMS_RTIKey MATCH 'ab*c'`
	The dot ('.') character separates a sequence of mandatory substrings. Each '.' In the pattern must correspond to a '.' in the field value.	`JMS_RTIKey MATCH 'Equities.*.A*'`
	Square brackets ('[', ']') surround a set of characters, one of which must occur in the field value.	`JMS_RTIKey MATCH 'hell[oap]'`
	Two characters within square brackets and separated by a hyphen indicate an inclusive range.	`JMS_RTIKey MATCH 'hell[a-z]'`
	The comma ',' character delimits a list of patterns. The filter passes if the field value matches any pattern in the list.	`JMS_RTIKey MATCH 'Equities.*.A[A-Z]*,Equities.NYSE.G[A-Z]*'`
'BETWEEN', 'NOT BETWEEN'	Message field value is within a specified range.	`JMS_RTIKey BETWEEN 'a' AND 'z'`
'AND', 'OR'	Multiple conditions can be combined together using these keywords.	`(JMSCorrelationID = 'foo') AND (JMS_RTIKey MATCH 'Equities.*.A*')`

5.3.1.2 Filtered Fields

Your application can filter on the following message header fields and properties using message selectors:

- **JMSCorrelationID**
- **JMSType**
- JMSReplyTo[1]
- **JMS_RTIKey**[2]

The *RTI Event Processing* system component can perform much more extensive filtering, as well as correlate events across multiple technologies, including database events, file modifications, *RTI Data Distribution Service*, IBM MQ, and so forth. For more information about *RTI Event Processing*, please contact your RTI account representative.

5.3.2 Advanced: Time-Based Filtering

In some data streams, it's not important for your application to get every message if the values in those messages change very quickly. For example, the values may be displayed for human interaction, and there is a limit to how fast that person can understand and react to them. Or the messages may contain frames of a video stream, and you need to adapt the frame rate to match the capabilities of different display devices.

RTI Message Service addresses these and similar use cases with time-based filters specifying a "minimum separation": the minimum time duration that should elapse in between messages. If messages arrive faster than this rate, the middleware will discard intermediate messages. For this reason, an application specifying a time-based filter is requesting something less than strict reliability; this intermediate level of reliability is not appropriate for all applications.

Time-based filters are specified in an application's configuration file.

1. For more information about these message header fields, see the JMS specification: http://java.sun.com/javaee/5/docs/api/javax/jms/Message.html.
2. For more information about keys and this property, see Advanced: Keyed Topics for Real-Time Performance and Scalability (Section 3.4) or Chapter 8: Scalable High-Performance Applications: Keys.

Example:

```
<topic name="Example Topic">
    <consumer_defaults>
            <time_based_filter>
                    <minimum_separation>
                            <sec>3</sec>
                            <nanosec>30</nanosec>
                    </minimum_separation>
            </time_based_filter>
    </consumer_defaults>
</topic>
```

Part 2:Advanced Concepts

This section includes advanced material that will help you tune your application for high-performance scalability.

- Chapter 6: Scalable High-Performance Applications: Message Reliability
- Chapter 7: Scalable High-Performance Applications: Durability and Persistence for High Availability
- Chapter 8: Scalable High-Performance Applications: Keys

Chapter 6 Scalable High-Performance Applications: Message Reliability

To build scalable, high-performance applications, it's not enough to simply be familiar with **send** and **receive** methods. You need to understand the requirements of your data stream and know how to declare those requirements to the middleware so that it can deliver to you the semantics and performance you need and expect.

This chapter provides advanced information about configuring your application for the appropriate levels of reliability and determinism to bridge the gap from simple examples to production applications. It assumes that you have already read the Getting Started Guide and executed the tutorial there. It also assumes that you have read and understood the more-general information contained in the earlier chapters of this manual.

This chapter is organized as follows:

6.1 Introduction to Reliability

RTI Message Service supports a broader range of reliability contracts than any other JMS implementation.

Best-effort	Messages that are dropped by the network or that arrive out of order will not be delivered to the application. This capability is an extension to the JMS specification.
Filtered Reliability	Messages may be sent either best-effort or reliably, but will only be delivered to the application if they pass certain content-based or minimum-separation-based filters specified by the application. Content-based filters are described by the JMS-standard "message selector" parameter that can be provided when creating a subscriber, although RTI Message Service does provide some advanced filtering features that extend the SQL grammar supported by other JMS vendors. Time-based filters are an extension to the JMS specification. Content- and time-based filters can optionally be combined with space- and/or time-windowed reliability. Message filtering is described in the Subscribing to Messages chapter of this manual and is not discussed further here.
Space-Windowed Reliability	Messages are sent reliably, but will not be delivered to the application if more than an application-specified number of subsequent messages have ar-rived by the time the first method is eligible for de-livery. Space-windowed reliability is an extension to the JMS specification, and can optionally be combined with time-windowed reliability and/or filtered re-liability.
Time-Windowed Reliability	Messages are sent reliably, but will not be delivered to the application if the elapsed time between when the message was sent and when it is eligible for delivery exceeds an application-specified threshold. This capability is part of the JMS specification and corresponds to the getTimeToLive and setTimeToLive methods on a MessageProducer. It can optionally be combined with space-windowed reliability and/or filtered reliability.
Strict Reliability	Messages are sent reliably and are guaranteed to be delivered to the application, provided that no net-work component or application fails before then. Persisting messages to provide high availability in the face of such failures is described in the chapter Message Durability and Persistence for High Avail-ability.

By default, if your configuration omits all reliability configuration, message producers are configured to support either best-effort or reliable consumers, and consumers are configured for best-effort communication.

Regardless of whether delivery is reliable or best-effort, *RTI Message Service* will never deliver messages out of order. If delivery is reliable, out-of-order messages will be queued internally until all previous messages have arrived, and then the messages will be delivered in the order in which they were sent. If delivery is best-effort, messages will be delivered to the application as soon as they arrive. If a message arrives that was sent before another message that was already delivered to the application, that message will be dropped by the middleware.

6.1.1 QoS Policies

The reliability contracts described above are configured with several inter-related QoS policies. These policies can be configured on a per-topic basis, in which case they will apply in identical fashion to publishers and subscribers of that topic, or they may be defined independently for publishers and subscribers. These policies include:

- **Reliability**—The level of reliability is defined by a reliability **kind**, which must be one of BEST_EFFORT_RELIABILITY_QOS or RELIABLE_RELIABILITY_QOS.

  As described above, the default reliability **kind** for publishers is RELIABLE_RELIABILITY_QOS, which indicates the most stringent contract supported by that publisher; it can communicate with either reliable or best-effort subscribers.

- **History**—This policy defines the space limitations on reliability; it controls whether the middleware should deliver every message, retain only the last value, or something in be-tween. The two most important fields it defines are the history **kind** and the history **depth**. The **kind** indicates whether reliability will be space-windowed or not; it must take one of the values KEEP_LAST_HISTORY_QOS or KEEP_ALL_HISTORY_QOS. The **depth** applies when the kind is KEEP_LAST_HISTORY_QOS and indicates the number of previous messages to retain.

- **Lifespan**—The lifespan QoS policy defines a **time_to_live** that limits the maximum amount of time that the middleware will maintain sent messages; it governs time-windowed reliability. With respect to current subscribers, it is applicable in two cases:

If the reliability **kind** is RELIABLE_RELIABILITY_QOS, and a message arrives out of order (that is, before a message that was sent before it), the middleware must retain it internally until all earlier messages arrive so that all messages can be delivered in order without losses. The **time_to_live** governs how long the middleware will wait for these earlier messages to arrive.

If a subscribing application is receiving messages synchronously, some time could elapse in between when the message arrives from the network and is ordered by the middleware and when the application issues the next call to a **receive()** method. If this elapsed time exceeds the **time_to_live**, the middleware will discard any obsolete messages and deliver the first available message whose **time_to_live** has not expired.

The default time to live is infinite.

- **Resource Limits**—For efficiency, each *RTI Message Servic*e publisher and subscriber pre-allocates a pool of messages and re-uses the messages in that pool as, on the publisher side, messages are sent, and on the subscriber side, they arrive over the network. The resource limits policy governs the size of this pool. Use this policy to control the middle-ware's memory use as it carries out its reliability contract and to throttle the middleware to maintain high performance when producers and consumers operate at different rates.

  The most important member of the resource limits policy is **max_messages**. This parameter, which is unlimited by default, can be important when the history kind is set to KEEP_ALL_HISTORY_QOS. Consider the case where a producer configured to keep all unacknowledged historical messages, and that producer continuously sends messages faster than its subscribers can keep up. By default, because max_messages is infinite, the producer will continue to enqueue sent messages indefinitely, getting further ahead and growing its memory footprint as it goes. The max_messages parameter provides a mechanism to limit memory use and throttle the sender. When this limit is reached, a send invocation will block until the producer receives sufficient acknowledgements to make space available in its history.

6.1.2 JMS Acknowledgement Modes

RTI Message Service supports two of the JMS-standard acknowledgement modes:

- Session.AUTO_ACKNOWLEDGE. The specification indicates that the implementation automatically acknowledges receipt of a message immediately after the return of either (a) **MessageConsumer.receive()** or (b) **MessageListener.onMessage()**.

- Session.DUPS_OK_ACKNOWLEDGE. The specification indicates that the implementation may "lazily" acknowledge the delivery of messages, even if that behavior would result in the delivery of some duplicate messages if the JMS provider fails.

These definitions assume an architecture that is poorly suited for high-performance and/or real-time distributed applications.

- In AUTO mode, it assumes that message delivery state will be persisted to permanent storage before any messages are acknowledged, because this is the only way that implementations can assure that there will be no duplicate delivery in the event of a failure. The introduction of a database into the critical message delivery path can dramatically impact both latency and determinism.
- In AUTO mode, it couples message acknowledgement to application calls to **MessageConsumer.receive()**. Such an architecture can substantially degrade performance and is ill-suited to modern multi-core, multi-threaded system architectures.
- In DUPS_OK mode, it makes no guarantees with respect to causality or timeliness, making the implementation's behavior difficult to understand or predict.

The acknowledgement behavior of *RTI Message Service* provides the consistent, deterministic, efficient behavior high-performance applications need. It is the same, regardless of the acknowledgement mode.

1. The middleware reads messages from network sockets using one or more internal threads.
2. When an incoming message is ready to be delivered to the application-that is, when the messages of all previous sequence numbers have been accounted for, so that there are no "holes"- the middleware will deliver it to the appropriate *MessageListener*, if one is in-stalled.
3. Immediately afterwards, the middleware will send an acknowledgement.

This behavior is consistent with AUTO_ACKNOWLEDGE_MODE with respect to asynchronous deli-very in the case where the messaging infrastructure does not fail. It is consistent with DUPS_OK_ACKNOWLEDGE_MODE with respect to synchronous delivery in that acknowledgement is not dependent on a call to **receive()**.

If there is no failure in the messaging infrastructure, an application will never observe duplicate messages. An application can also prevent duplicate delivery in the face of failures, but this prevention is not related to the message acknowledgement mode. Instead, you should configure the middleware for durable consumer state; see Chapter 7: Scalable High-Performance Applications: Durability and Persistence for

High Availability for more information about this capability.

6.1.3 Message Loss and Rejection Notification

Regardless of your reliability configuration, your application can detect when a message has been lost. Note that, in the case of best-effort delivery, the middleware will not be able to detect the loss until and unless a subsequent data message does get through.

Table 6.1 **Notification Type: StatusNotifier.MESSAGE_LOST_NOTIFICATION_TYPE**

A subscriber has detected that a message has been lost and will not be delivered.

Attribute Name	Attribute Type	Description
totalCount	int	The total number of times that the subscriber has detected a message loss since it was created.
totalCountChange	int	The change to the totalCount attribute since the last time this status was queried. If your application receives status notifications via a listener callback, this number will generally be 1. If your application polls for status changes, it may be any integer greater than or equal to 1.

When resource usage has been limited on the consumer side, and the consumer exhausts those limits, it will stop enqueuing incoming messages. Two things will occur at that point: First, the consumer will notify the application if subsequent messages arrive that it has to reject. Second, because the consumer will stop acknowledging sent messages, the producer will eventually exhaust its own resource limits and block on sending new messages. This will give the consumer time to process its pending messages, resume acknowledgements, and thereby allow the producer to proceed.

Rejected messages are not lost; they can be resent when the consumer catches up.

When resource usage has been limited on the consumer side, and the consumer exhausts those limits, it will stop enqueuing incoming messages. Two things will occur at that point: First, the consumer will notify the application if subsequent messages arrive that it has to reject. Second, because the consumer will stop acknowledging sent messages, the producer will eventually exhaust its own resource limits and block on sending new messages. This will give the consumer time to process its pending messages, resume acknowledgements, and thereby allow the producer to proceed.

Rejected messages are not lost; they can be resent when the consumer catches up.

6.2 Best-Effort Delivery

For many data streams, reliability is a strong requirement. For others, it is not, and sending data best-effort can reduce latency and improve determinism. If one of the following conditions applies to your data streams, you may want to consider best-effort delivery.

- ❑ It is more important for your messages to arrive in a timely manner than for every message to arrive. For example, in a streaming audio or video application, the constancy of the streaming rate affects the user experience much more than an occasional dropped frame or sample.
- ❑ You're enforcing time or space limits on the reliability protocol, and the rate at which new messages arrive is large compared to the rate at which messages are dropped by the network. If a message is lost, a new one will arrive very shortly that supersedes the one that was lost.

When a data stream is configured for best-effort message delivery, the amount of metadata on the network can be substantially reduced: the publisher will not inform subscribers of which messages it has previously sent, and subscribers will not send acknowledgements to their publisher(s).

Note: The standard JMS API does not provide a way to indicate bet-effort reliability; this capability is an extension to JMS. In order to keep application code portable, *RTI Message Service* takes its reliability configuration from its configuration file, not the acknowledgement mode specified in code. If the configuration file specifies best-effort delivery, an acknowledgement mode of AUTO_ACKNOWLEDGE or DUPS_OK_ACKNOWLEDGE will be ignored; the subscriber will send no acknowledgements at all.

Example: Deliver the latest message as soon as it arrives without repairing losses

Your application may need only the latest message on a topic. In the event that a message is lost, you expect a refreshed state to arrive soon, so there's no need to repair the loss. This configuration is common for certain streaming sensor data use cases.

If this is the case, there is no configuration necessary.

Example: Deliver all messages; some losses are acceptable

You don't want to middleware to proactively discard any messages that were sent, but if a message is occasionally dropped in transit, there is no need for the publisher to resend it. This configuration may be valid for streaming media.

```
<topic name="Example Topic">
    <history>
           <kind>KEEP_ALL_HISTORY_QOS</kind>
    </history>
</topic>
```

6.3 Strictly Reliable Delivery

If a data stream is configured for strictly reliable delivery, the middleware will make every effort to provide all sent messages to all existing subscribers. This configuration places the greatest emphasis on getting a given message through and the least on freeing resources to handle the next message.

Strict reliability is often the preferred configuration for "command" or "alarm" topics or other data streams in which messages are sporadic and the delivery of each is critical. It is also important for data streams in which message represent changes with respect to a previous state, since the loss of any intermediate values may cause the recipient to misinterpret the subsequent values it receives.

Example: Reliably deliver every message to all current subscribers

Strict reliability requires not only reliability, but keep-all history.

```
<topic name="Example Topic">
    <reliability>
           <kind>RELIABLE_RELIABILITY_QOS</kind>
    </reliability>
    <history>
           <kind>KEEP_ALL_HISTORY_QOS</kind>
    </history>
</topic>
```

Example: Reliably deliver every message to all current subscribers, but force the producer to block if it accumulates 50 unacknowledged messages

Strict reliability requires not only reliability, but keep-all history. To throttle the publisher and limit its memory use, limit its resources as well.

```
<topic name="Example Topic">
    <reliability>
           <kind>RELIABLE_RELIABILITY_QOS</kind>
    </reliability>
    <history>
           <kind>KEEP_ALL_HISTORY_QOS</kind>
    </history>
    <producer_defaults>
           <resource_limits>
                  <max_messages>50</max_messages>
           </resource_limits>
    </producer_defaults>
</topic>
```

Example: Reliably deliver every message to all current subscribers, but only allow the consumer to accumulate 50 un-received messages

Strict reliability requires not only reliability, but keep-all history. However, if the consumer is receiving messages synchronously, and the application fails to call receive as fast as new messages arrive, it is important to limit the ability of the consumer to enqueue (and acknowledge) incoming messages. Otherwise, its memory footprint will continue to grow, and its producer(s) will continue to send new data.

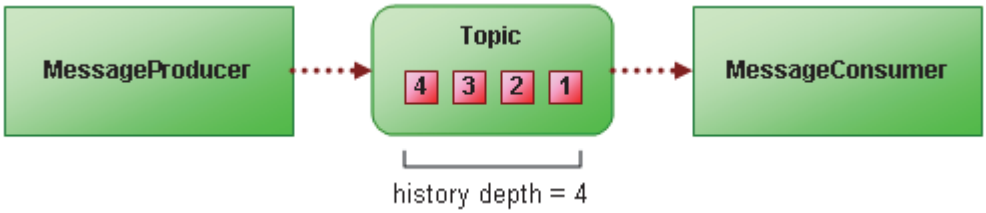
```
<topic name="Example Topic">
    <reliability>
           <kind>RELIABLE_RELIABILITY_QOS</kind>
    </reliability>
    <history>
           <kind>KEEP_ALL_HISTORY_QOS</kind>
    </history>
    <consumer_defaults>
           <resource_limits>
                  <max_messages>50</max_messages>
           </resource_limits>
    </consumer_defaults>
</topic>
```

6.4 Windowed Reliability

For many data streams, it is not actually required for subscribers to receive every message. What is required is for them to receive every recent message, where "recent" is defined by some time period and/or number of messages. This approach provides a balance between data determinism, with which subscribers can be assured of receiving sent messages, and time determinism, with which subscribers can count on more stable latencies and CPU loads.

6.4.1 Space-Windowed Reliability

Space-windowed reliability relies on three settings: a reliability kind of RELIABLE_RELIABILITY_QOS, a history kind of KEEP_LAST_HISTORY_QOS (the default), and a finite history depth (the default is 1).



Example: Reliably deliver the message that was sent most recently

This configuration is often used for topics that communicate the current state of some entity: whenever that state changes, a new message is sent out, and this message replaces any previous state announcement. Reliability must be activated; the history settings—keep the last single message—are left at their default values.

```
<topic name="Example Topic">
    <reliability>
           <kind>RELIABLE_RELIABILITY_QOS</kind>
    </reliability>
</topic>
```

Example: Reliably deliver the last five messages

```
<topic name="Example Topic">
    <reliability>
           <kind>RELIABLE_RELIABILITY_QOS</kind>
    </reliability>
    <history>
        <!-- May be omitted: <kind>KEEP_LAST_HISTORY_QOS</kind> -->
        <depth>5</depth>
    </history>
</topic>
```

6.4.2 Time-Windowed Reliability

Time-windowed reliability relies on two settings: a reliability kind of RELIABLE_RELIABILITY_QOS and a finite lifespan time to live (the default is infinite; that is, there is no time window).

The time to live can be specified in the configuration file, programmatically in application code, or both.

Example: Specify a time to live of 1 second in the configuration file

```
<topic name="Example Topic">
    <reliability>
           <kind>RELIABLE_RELIABILITY_QOS</kind>
    </reliability>
    <lifespan>
           <time_to_live>
                  <sec>1</sec>
                  <nanosec>0</nanosec>
           </time_to_live>
    </lifespan>
</topic>
```

Example: Specify an infinite time to live in the configuration file

It is typically not necessary to specify this configuration explicitly, because it is the default. But it is described here, because the syntax is slightly different in XML than in Java. In the configuration file, the minimum time to live is one nanosecond; a value of zero will result in an error.

```
<topic name="Example Topic">
    <reliability>
           <kind>RELIABLE_RELIABILITY_QOS</kind>
    </reliability>
    <lifespan>
           <time_to_live>
                  <sec>DURATION_INFINITE_SEC</sec>
                  <nanosec>DURATION_INFINITE_NSEC</nanosec>
           </time_to_live>
    </lifespan>
</topic>
```

Example: Specify a time to live of 1 second in application code

```
MessageProducer myPublisher = …;
try {
    myPublisher.setTimeToLive(1000); // milliseconds
} catch (JMSException jx) {
    // Respond to exception...
}
```

Example: Specify an infinite time to live in application code

It is typically not necessary to specify this configuration explicitly, because it is the default. But it is described here, because the syntax is slightly different in Java than in XML. In code, an infinite time to live is represented by the sentinel value 0.

```
MessageProducer myPublisher = …;
try {
    myPublisher.setTimeToLive(0);
} catch (JMSException jx) {
    // Respond to exception...
}
```

6.4.3 Complex Reliability Examples

The various reliability tuning mechanisms described in this chapter can be combined to achieve more fine-grained goals. Because these goals can be achieved within the mid-

dleware, the amount of application code you have to write is reduced, and performance can be enhanced, because the middleware can optimize delivery based on your needs.

Example: Only the last 2 messages are relevant, and a given message is relevant for only 200 milliseconds

This example limits the scope of reliable delivery based on both time and space constraints.

```
<topic name="Example Topic">
    <reliability>
           <kind>RELIABLE_RELIABILITY_QOS</kind>
    </reliability>
    <history>
        <!-- May be omitted: <kind>KEEP_LAST_HISTORY_QOS</kind> -->
        <depth>2</depth>
    </history>
    <lifespan>
        <time_to_live>
               <sec>0</sec>
               <nanosec>200000000</nanosec>
         </time_to_live>
    </lifespan>
</topic>
```

Example: Strict reliability, but throttle publisher and subscriber

Limit memory usage to throttle fast producers and prevent ballooning resource usage.

```
<topic name="Example Topic">
    <reliability>
           <kind>RELIABLE_RELIABILITY_QOS</kind>
    </reliability>
    <history>
           <kind>KEEP_ALL_HISTORY_QOS</kind>
    </history>
    <resource_limits>
           <max_messages>50</max_messages>
    </resource_limits>
</topic>
```

Chapter 7 Scalable High-Performance Applications: Durability and Persistence for High Availability

To build scalable, high-performance applications, it's not enough to simply be familiar with **send** and **receive** methods. You need to understand the requirements of your data stream and know how to declare those requirements to the middleware so that it can deliver to you the semantics and performance you need and expect.

This chapter provides advanced information about configuring your application for the appropriate level of data availability to bridge the gap from simple examples to production applications. It assumes that you have already read the Getting Started Guide and executed the tutorial there. It further assumes that you have read and understood the more-general information contained in the earlier chapters of this manual.

Durability and persistence are closely related to reliability. If you have not yet read the previous chapter on reliability and deterministic delivery, it is recommended that you do so before continuing.

This chapter is organized as follows:

7.1 Introduction to Durability and Persistence

The concepts of durability and persistence build on that of reliability to support a range of critical application requirements. *RTI Message Service* provides one of the most comprehensive capabilities in the industry, allowing you to achieve the right balance between latency, determinism, and high availability on a per-topic basis.

- **Basic reliability**—Delivery of messages to current subscribers, provided that no component of the messaging infrastructure fails before the message can be acknowledged. The *RTI Message Service* reliability model, including its most common configuration use cases, is described in Chapter 6: Scalable High-Performance Applications: Message Reliability of this manual.
- **Enhanced availability for late-joining subscribers**—Delivery of historical messages to subscribers that join the network after those messages were sent originally, provided that the publisher is still operational.
- **Enhanced reliability in the face of publisher restarts**—Delivery of historical messages to both current and late-joining subscribers, regardless of whether the original publisher shut down, provided that it starts up again somewhere.
- **Enhanced reliability in the face of subscriber restarts**—Avoiding duplicate delivery of historical messages to subscribers if they shutdown and restart, either on the same node or elsewhere on the network.
- **Enhanced availability when the publisher is not running**—Delivery of historical messages to current and late-joining subscribers when the publisher is not running, either before the publisher restarts or in the case where the publisher does not restart at all.
- **Enhanced availability in the face of persistence service failure**—Redundancy and load balancing in the persistence layer of the messaging infrastructure to assure message availability, even in the face of multiple failures.

RTI Message Service supports this extensive list of requirements by combining four fundamental capabilities.

- **Message Durability**—*RTI Message Service* exposes a comprehensive set of message durability levels that allow applications to configure message streams for no historical message retention, for retention of all messages persistently in a database for an indefinite period, or anything in between. This level of durability is configured simply and declaratively in the application's configuration file.
- **Durable Producer History**—A *MessageProducer* can be configured to persist its cache of historical messages to a database so that it can survive shutdowns, crashes and restarts. When an application restarts, each *MessageProducer* configured with a durable history automatically loads all the data in its history cache from disk and can carry on sending and repairing messages as if it had never stopped executing. To the rest of the system, it will appear as if the *MessageProducer* had been temporarily disconnected from the network and then reappeared.

- **Durable Consumer State**—A *MessageConsumer* can be configured to persist its message acknowledgement state such that, if it shuts down or fails and subsequently restarts, it will not deliver any duplicate messages to the application. When an application restarts, each *MessageConsumer* that has been configured to have durable state automatically loads that state from disk and can carry on receiving data as if it had never stopped executing. In fact, any publishers with which it communicates will not resend any messages that were acknowledged before the failure, avoiding unnecessary network utilization.
- **RTI Persistence Service**—The *RTI Persistence Service* is a process that runs on your network and provides access to sent messages any time the original publishers of those messages are not running. Any number of instances of this service can be configured for redundancy and/or load balancing to provide the level of availability, assurance, and performance your system requires.

These features can be configured separately or in combination. To use durable producer history or durable consumer state, or to permanently persist messages using the *RTI Persistence Service*, you need a relational database, which is not included with *RTI Message Service*. Supported databases are listed in the Release Notes.

To understand how these features interact we will examine the behavior of the system using the following scenarios:

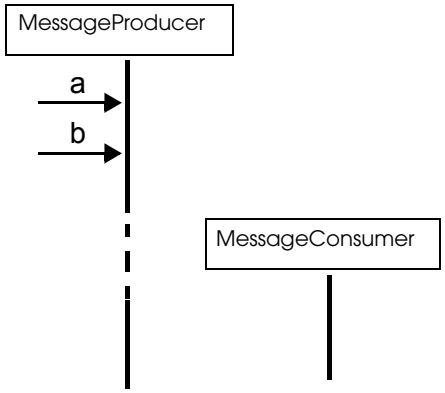
- Scenario 1. A MessageConsumer Joins after a MessageProducer Restarts (Durable Producer History) (Section 7.1.1)
- Scenario 2: A MessageConsumer Restarts While MessageProducer Stays Up (Durable Consumer State)
- Scenario 3. A MessageConsumer Joins after the MessageProducer Leaves the Network (Durable Data)

7.1.1 Scenario 1. A MessageConsumer Joins after a MessageProducer Restarts (Durable Producer History)

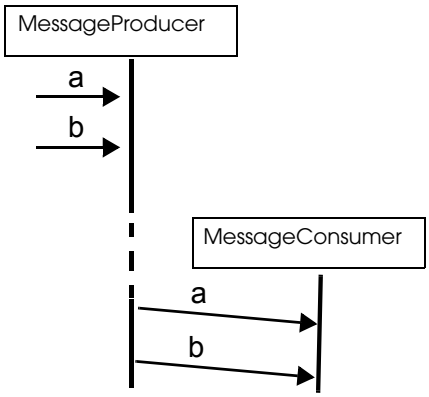
In this scenario, an application joins the network, creates a *MessageProducer* and publishes some data, then the *MessageProducer* shuts down (gracefully or due to a fault). The *MessageProducer* restarts and a *MessageConsumer* joins the domain. Depending on whether the *MessageProducer* is configured with durable history, the late-joining *MessageConsumer* may or may not receive the data published already by the *MessageProducer*

before it restarted. This is illustrated in Figure 7.1. For more information, see Durable Producer History (Section 7.4).

Figure 7.1 **Durable Producer History**



Without Durable Producer History:
the late-joining MessageConsumer will not receive data (a and b) that was published before the MessageProducer restarts.



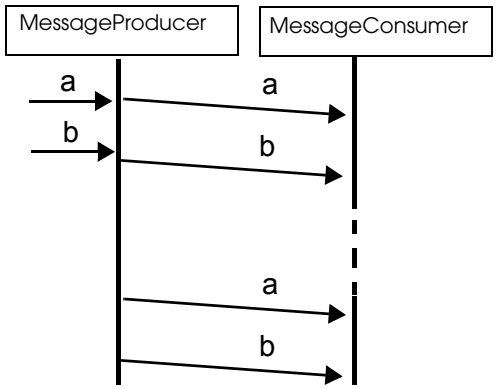
With Durable Producer History:
the restarted MessageProducer will recover its history and deliver its data to the late-joining MessageConsumer

7.1.2 Scenario 2: A MessageConsumer Restarts While MessageProducer Stays Up (Durable Consumer State)

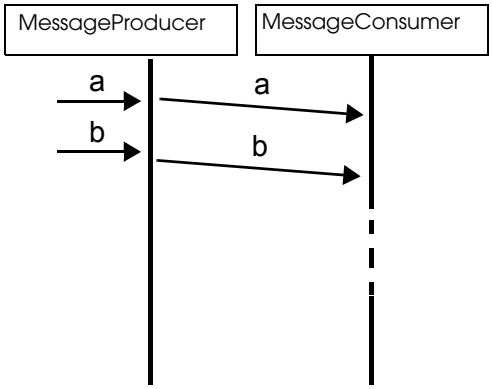
In this scenario, two applications join a network; one creates a *MessageProducer* and the other a *MessageConsumer* on the same Topic. The *MessageProducer* publishes some data ("a" and "b") that is received by the *MessageConsumer*. After this, the *MessageConsumer* shuts down (gracefully or due to a fault) and then restarts—all while the *MessageProducer* remains present in the domain.

Depending on whether the *MessageConsumer* is configured with Durable Consumer State, the *MessageConsumer* may or may not receive a duplicate copy of the data it received before it restarted. This is illustrated in Figure 7.2. For more information, see Durable Consumer State (Section 7.5) .

Figure 7.2 **Durable Consumer State**



Without Durable Consumer State: the MessageConsumer will receive the data that was already received before the restart.



With Durable Consumer State: the MessageConsumer remembers that it already received the data and does not request it again.

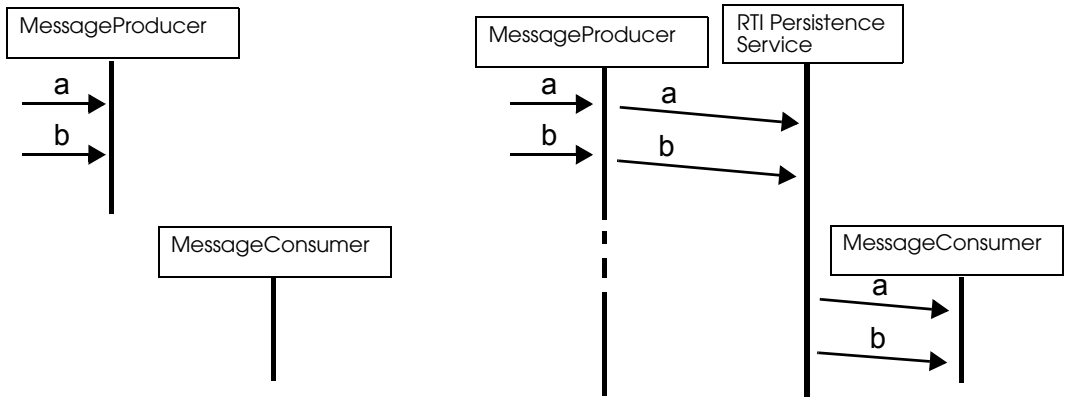
7.1.3 Scenario 3. A MessageConsumer Joins after the MessageProducer Leaves the Network (Durable Data)

In this scenario, an application joins the network, creates a *MessageProducer*, publishes some data on a *Topic* and then shuts down (gracefully or due to a fault). Later, a *MessageConsumer* joins the domain and subscribes to the data. *RTI Persistence Service* is running.

Depending on whether message durability is enabled for the Topic, the *MessageConsumer* may or may not receive the data previous published by the *MessageProducer*. This is illustrated in Figure 7.3. For more information, see Message Durability (Section 7.2).

This third scenario is similar to Scenario 1. A MessageConsumer Joins after a MessageProducer Restarts (Durable Producer History) except that in this case the *MessageProducer* does not need to restart for the *MessageConsumer* to get the data previously written by the *MessageProducer*. This is because *RTI Persistence Service* acts as an intermediary that stores the data so it can be given to late-joining *MessageConsumers*.

Figure 7.3 **Durable Data**



Without Durable Data:
the late-joining MessageConsumer will not receive data (a and b) that was published before the MessageProducer quit.

With Durable Data:
RTI Persistence Service remembers what data was published and delivers it to the late-joining MessageConsumer.

7.2 Message Durability

Because the publish-subscribe paradigm of *RTI Message Service* is connectionless, applications can create publications and subscriptions in any way they choose. As soon as a matching pair of *MessageProducer* and *MessageConsumer* exists, then data published by the *MessageProducer* will be delivered to the *MessageConsumer*. However, a *MessageProducer* may publish data before a *MessageConsumer* has been created. For example, before you subscribe to a magazine, there may have been previous issues published.

RTI Message Service allows applications to control whether or not, and how, messages are stored by publishers for subscribers that are found after messages were initially written. The analogy is for a new magazine subscriber to ask for issues that were published in the past.

This capability also addresses the needs of subscribers that may require access to data even after the original publisher has shut down. Just like it is possible to access historical magazine issues, even after the publishing house has closed its doors, so too can a

subscriber access historical messages after their original publishers have halted, whether intentionally or due to a fault.

The following levels of durability address these use cases:

- **VOLATILE**: The middleware does not need to keep any message on behalf of any subscriber that is unknown to the publisher at the time the message is written. Once a message has been acknowledged by all current subscribers, it can be removed from the publisher's cache. If the publisher fails before the message is acknowledged, the message could be lost. *This is the highest-performing configuration and the default.*
- **TRANSIENT_LOCAL**: The publisher will keep some number of historical messages so that they can be delivered to any potential late-joining subscribers. However, the messages will not be persisted outside of the publisher itself, so if it shuts down or fails before the message is acknowledged, the message could be lost.
- **TRANSIENT**: The messaging infrastructure will keep some number of historical messages so that they can be delivered to any potential late-joining subscribers. These messages will be retained externally to the original publisher, so that if the publisher fails, the message will remain available. However, the messages will not be persisted to permanent storage, so if the publisher and the service both fail, the message could be lost.
- **PERSISTENT**: Messages are kept in permanent storage external to the original publisher so that they can outlive multiple failures or even a full system restart.

The TRANSIENT and PERSISTENT levels require a relational database, connected to the messaging infrastructure using Durable Producer History and/or one the *RTI Persistence Service*.

7.2.1 QoS Policies

Durability is configured primarily with the durability QoS policy. This policy includes a durability **kind** that identifies the level of durability as described above: its value must be one of VOLATILE_DURABILITY_QOS, TRANSIENT_LOCAL_DURABILITY_QOS, TRANSIENT_DURABILITY_QOS, or PERSISTENT_DURABILITY_QOS.

The durability QoS policy is related to other policies:

- Since best-effort subscribers will not inform publishers of messages they have not received, durability kinds greater than VOLATILE_DURABILITY_QOS are only effective for messages that are sent reliably.

- ❏ The number of historical messages that will be stored is defined by the history depth.
- ❏ If the history is configured to keep all historical messages, and the durability is greater than volatile, it is important to configure finite resource limits to avoid increasing your memory footprint indefinitely with each message sent.

For more information about configuring the reliability, history, and resource limits QoS policies, see Chapter 6: Scalable High-Performance Applications: Message Reliability.

The durability QoS policy can be set on publisher and/or subscriber. If it is set on both, the level of durability offered by the publisher must be greater than or equal to that requested by the subscriber. If that condition does not hold, the publisher and subscriber will not be able to communicate. For example, a volatile publisher will not communicate with a persistent subscriber.

Example: The 10 most-recent messages are relevant to subscribers; these will be retained locally for late-joining subscribers

This is a variation of the space-windowed reliability use case described in Space-Windowed Reliability (Section 6.4.1). That example is enhanced with transient-local durability.

```
<topic name="Example Topic">
    <reliability>
            <kind>RELIABLE_RELIABILITY_QOS</kind>
    </reliability>
    <history>
            <!-- May be omitted: <kind>KEEP_LAST_HISTORY_QOS</kind> -
->
            <depth>10</depth>
    </history>
    <durability>
            <kind>TRANSIENT_LOCAL_DURABILITY_QOS</kind>
    </durability>
</topic>
```

Example: Strictly-reliable publisher provides its local cache of historical messages to late-joining subscribers

This is a variation of the strict reliability use case described in the Message Reliability chapter. That example is enhanced with transient-local durability. When adding durability to a strictly reliable data stream, it's important to limit resource usage to prevent the publisher's memory footprint from growing indefinitely with each message it sends.

```
<topic name="Example Topic">
    <reliability>
           <kind>RELIABLE_RELIABILITY_QOS</kind>
    </reliability>
    <history>
           <kind>KEEP_ALL_HISTORY_QOS</kind>
    </history>
    <durability>
           <kind>TRANSIENT_LOCAL_DURABILITY_QOS</kind>
    </durability>
    <resource_limits>
           <max_messages>25</max_messages>
    </resource_limits>
</topic>
```

7.2.2 Configuring External Durability with RTI Persistence Service

Levels of durability greater than transient-local dictate that messages be stored externally to the publisher itself in order to maintain data availability in the event that the original publisher shuts down. If in your distributed system, the publisher is expected to immediately restart and resume operation, you may find that Durable Producer History meets your needs. If, however, you require that data remain available while the publisher is not running, you will need to deploy the *RTI Persistence Service*.

For more information on *RTI Persistence Service*, please see:

- ❏ Chapter 20: Introduction to RTI Persistence Service
- ❏ Chapter 21: Configuring RTI Persistence Service
- ❏ Chapter 22: Running RTI Persistence Service

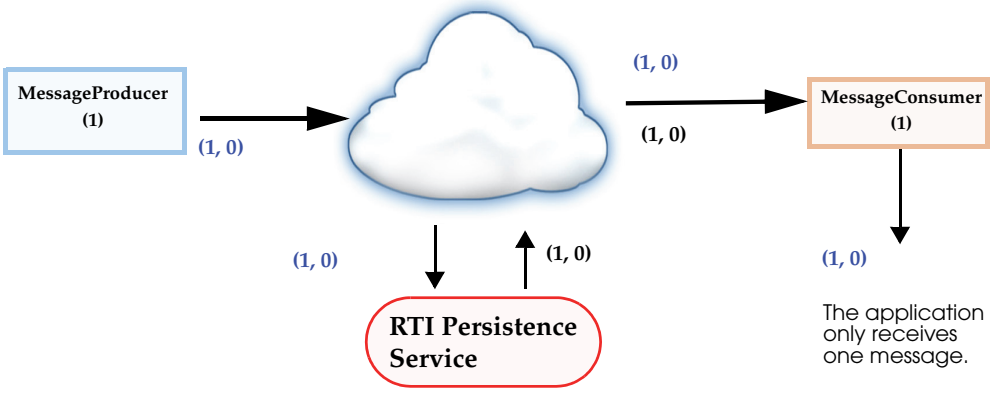
RTI Persistence Service can be configured to operate in PERSISTENT or TRANSIENT mode:

- **TRANSIENT mode** *RTI Persistence Service* will keep the received messages in memory. Messages published by a TRANSIENT *MessageProducer* will survive the *MessageProducer* lifecycle but will not survive the lifecycle of *RTI Persistence Service* (unless you are running multiple copies).
- **PERSISTENT mode** *RTI Persistence Service* will keep the received messages in a relational database. Messages published by a PERSISTENT *MessageProducer* will survive the *MessageProducer* lifecycle as well as any restarts of *RTI Persistence Service*.

Peer-to-Peer Communication:

By default, a PERSISTENT/TRANSIENT *MessageConsumer* will receive messages directly from the original *MessageProducer* if it is still alive. In this scenario, the *MessageConsumer* may also receive the same messages from *RTI Persistence Service*. Duplicates will be discarded at the middleware level. This peer-to-peer communication pattern is illustrated in Figure 7.4.

Figure 7.4 **Peer-to-Peer Communication**



Relay Communication:

A PERSISTENT/TRANSIENT *MessageConsumer* may also be configured to not receive messages from the original *MessageProducer*, but from the *RTI Persistence Service* only. This 'relay communication' pattern is illustrated in Figure 7.5. To use relay communication, set the **direct_communication** field in the durability Qos policy to false. A PERSISTENT/TRANSIENT *MessageConsumer* will receive all the information from *RTI Persistence Service*.

This model enhances assurance, in that all messages have been persisted before they are received, but at the cost of latency.

Figure 7.5 **Relay Communication**

Relay Communication Example:

The last 10 messages will be retained persistently (that is, in a relational database) by one or more instances of the *RTI Persistence Service*. Consumers will only accept messages relayed through a service instance; they will not communicate directly with the original message publisher.

```
<topic name="Example Topic">
    <reliability>
           <kind>RELIABLE_RELIABILITY_QOS</kind>
    </reliability>
    <history>
       <!-- May be omitted: <kind>KEEP_LAST_HISTORY_QOS</kind> -->
       <depth>10</depth>
    </history>
    <durability>
       <kind>PERSISTENT_DURABILITY_QOS</kind>
       <direct_communication>false</direct_communication>
    </durability>
</topic>
```

7.3 Identifying Persisted Data

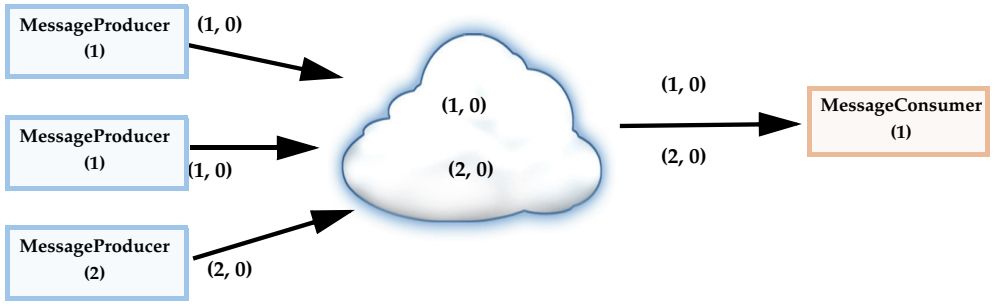
To configure Durable Producer History or Durable Consumer State, you will need to understand how the RTI messaging infrastructure internally identifies messaging objects, like producers and consumers, and the messages on which they operate.

Within the middleware, producers and consumers are associated with a 16-byte Globally Unique Identifier (GUID). This value is assigned by the middleware automatically; its uniqueness is critical for the correct operation of the middleware.

When a producer or consumer shuts down—whether intentionally or not—and comes back up, the middleware must be able to determine that what is physically a new messaging object with a unique GUID is *logically* equivalent to another object previously in existence. In order to build this association, the middleware also associates a *virtual GUID* with each producer and consumer. This value can be set by an application; if it is not, it will be the same as the physical GUID.

Every message ever published by a *MessageProducer* in *RTI Message Service* is uniquely identified by a tuple (virtual GUID, sequence number). The sequence number is an 8-byte ordinal that orders the message with respect to the *MessageProducer*. If two producers with the same virtual GUID publish messages with the same sequence number, those messages will be considered duplicates of the same message, and only one of them will be delivered to the application[1].

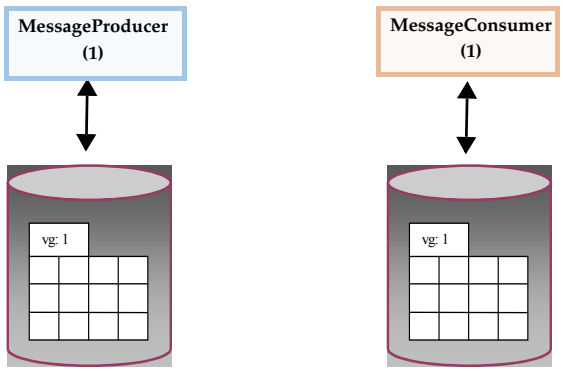
Figure 7.6 **Global Dataspace Changes**



1. To ensure correct behavior, we highly recommend that applications do not assign the same virtual GUIDs to logically different producers, as any loss of synchronization between then will result in non-identical messages being considered as duplicates and discarded.

RTI Message Service uses virtual GUIDs to associate a persisted state (state in permanent storage) to messaging objects to support the Durable Producer History, Durable Consumer State, and *RTI Persistence Service* capabilities.

Figure 7.7 **History/State Persistence Based on the Virtual GUID**



Configuration of virtual GUIDs is described in more detail below.

7.4 Durable Producer History

The durability QoS policy controls whether or not, and how, published messages are stored by the *MessageProducer* application for *MessageConsumers* that are found after the messages were initially written. The messages stored by the *MessageProducer* constitute the *MessageProducer's* history.

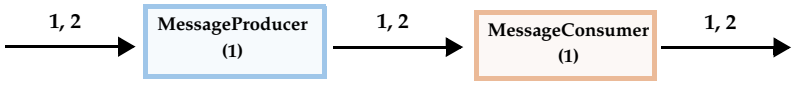
RTI Message Service provides the capability to make the *MessageProducer* history durable by persisting its content in a relational database. This makes it possible for the history to be restored when the *MessageProducer* restarts. See Chapter 5 in the Getting Started Guide for the list of supported relational databases.

The association between the history stored in the database and the *MessageProducer* is done using the virtual GUID.

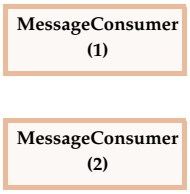
7.4.1 Durable Producer History Use Case

The following use case describes the durable writer history functionality:

1. A *MessageConsumer* receives two messages with sequence numbers 1 and 2 published by a *MessageProducer* with virtual GUID 1.

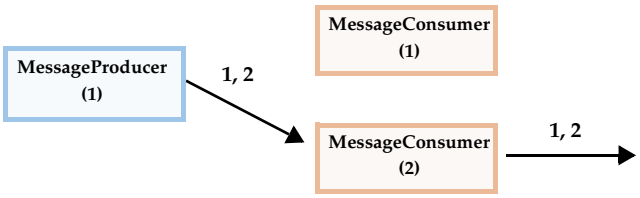


2. The process running the *MessageProducer* is stopped and a new late-joining *MessageConsumer* is created.



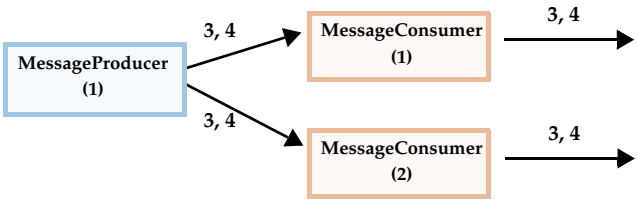
   The new *MessageConsumer* does not receive messages 1 and 2 because the original *MessageProducer* has been destroyed. If the messages must be available to late-joining *MessageConsumer* after the *MessageProducer* deletion, you can use *RTI Persistence Service*.

3. The *MessageProducer* is restarted using the same virtual GUID.



   After being restarted, the *MessageProducer* restores its history. The late-joining *MessageConsumer* will receive messages 1 and 2 because they were not received previously. The first *MessageConsumer* will not receive messages 1 and 2 because it already received them

4. The *MessageProducer* publishes two new messages.



The two new messages, with sequence numbers 3 and 4, will be received by both *MessageConsumers*.

7.4.2 How To Configure Durable Writer History

To configure durable producer history, use the property QoS policy of either an individual *MessageProducer* and or of a *ConnectionFactory*. A durable producer history property defined for a *ConnectionFactory* will be applicable to all the *MessageProducers* belonging to all *Connections* created from that *ConnectionFactory*, unless it is overwritten by a *MessageProducer*.

The property QoS policy is made up of key-value properties, and is configured as follows:

```
<property>
    <value>
            <element>
                    <name>property1</name>
                    <value>value1</value>
            </element>
            <element>
                    <name>property2</name>
                    <value>value2</value>
            </element>
    </value>
</property>
```

Table 7.1 lists the supported 'durable producer history' properties.

Table 7.1 **Durable Producer History Properties**

Property	Description
dds.data_writer.history.plugin_name	Must be set to **"dds.data_producer.history.odbc_plugin.builtin"** to enable durable producer history in the *MessageProducer*. This property is required.
dds.data_writer.history.odbc_plugin.dsn	The ODBC DSN (Data Source Name) associated with the database where the producer history must be persisted. This property is required.
dds.data_writer.history.odbc_plugin.driver	This property tells *RTI Message Service* which ODBC driver to load. If the property is not specified, *RTI Message Service* will try to use the standard ODBC driver manager library (UnixOdbc on UNIX/Linux systems, the Windows ODBC driver manager on Windows systems).
dds.data_writer.history.odbc_plugin.username	Configures the username/password used to connect to the database.
dds.data_writer.history.odbc_plugin.password	
dds.data_writer.history.odbc_plugin.shared	By default, *RTI Message Service* will create a single connection per DSN that will be shared across *MessageProducers*. To increase concurrency, a *MessageProducer* can be configured to create its own database connection by setting this property to 0 (false).
dds.data_writer.history.odbc_plugin.sample_cache_max_size	These properties configure the resource limits associated with the ODBC producer history caches. To minimize the number of accesses to the database, *RTI Message Service* uses a message cache. The initial size and the maximum size of these caches are configured using these properties. The producer resource limits, such as **max_messages**, defined in the resource limits QoS policy are used to configure the maximum number of messages that can be stored in the relational database.
dds.data_writer.history.odbc_plugin.sample_cache_init_size	

Table 7.1 **Durable Producer History Properties**

Property	Description
dds.data_writer.history.odbc_plugin.restore	This property indicates whether or not the persisted producer history must be restored once the *MessageProducer* is restarted. If this property is 0, the content of the database associated with the *MessageProducer* being restarted will be deleted. If it is 1, the *MessageProducer* will restore its previous state from the database content.
dds.data_writer.history.odbc_plugin.in_memory_state	This property determines how much state will be kept in memory by the ODBC producer history in order to avoid accessing the database. Activating this mode—setting this property to '1'—provides the best ODBC producer history performance. However, the restore operation will be slower, and the maximum number of messages that the producer history can manage is limited by the available physical memory. If the property is 0, all the state will be kept in the underlying database. In this mode, the maximum number of messages in the producer history is not limited by the physical memory available.

Example Configuration:

The virtual GUID's value is an array of 16 bytes. It can be specified in hexadecimal or unsigned decimal form.

```
<topic name="Example Topic">
    <producer_defaults>
        <protocol>
            <virtual_guid>
                <value>
  239,238,237,236,235,234,233,232,231,230,229,228,227,226,225,224
                </value>
            </virtual_guid>
        </protocol>
        <property>
            <value>
                <element>
                    <name>dds.data_writer.history.plugin_name
                    </name>
                    <value>
                        dds.data_writer.history.odbc_plugin.builtin
```

```
                    </value>
                </element>
                <element>
                    <name>dds.data_writer.history.odbc_plugin.dsn
                    </name>
                    <value>my user DSN</value>
                </element>
                <element>
                    <name>
                    dds.data_writer.history.odbc_plugin.driver
                    </name>
                    <value>my ODBC library</value>
                </element>
                <element>
                    <name>
                    dds.data_writer.history.odbc_plugin.shared
                    </name>
                    <value>1</value>
                </element>
            </value>
        </property>
    </producer_defaults>
</topic>
```

7.5 Durable Consumer State

Durable consumer state allows a *MessageConsumer* to locally persists its state and remember the data it has already received. When an application restarts, each *MessageConsumer* that has been configured to have durable consumer state automatically loads its state from disk. Data that was already received by the *MessageConsumer* before the restart will be suppressed so it is not sent over the network again.

RTI Message Service provides the capability to persist the state of a *MessageConsumer* in a relational database. This database is accessed using ODBC. See the Release Notes for the list of supported relational databases.

A message will be considered as received by a message consumer when:

- The **onMessage** method of the consumer's **MessageListener** returns. For example:

```
class MyListener implements MessageListener {
```

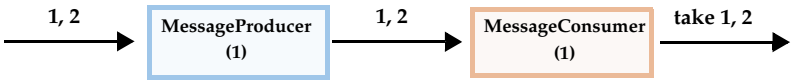
```
        public void onMessage(Message myMessage) {
               // Handle myMessage...
        }      // myMessage considered received
}
```

❑ When a **receive** method returns that message. For example:

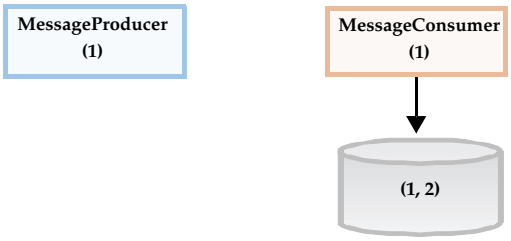
```
Message myMessage = myConsumer.receiveNoWait();
// myMessage considered received
// Handle myMessage...
```


7.5.1 Durable Consumer State Use Case

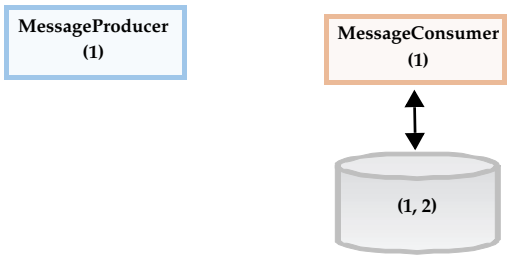
1. A MessageConsumer receives two messages with sequence numbers 1 and 2 published by a MessageProducer. The application receives those messages.



After the application has received each message, the MessageConsumer persists the state change.



2. The process running the *MessageConsumer* is stopped.
3. The *MessageConsumer* is restarted.



Because all the messages with sequence number smaller or equal than 2 were considered as received, the consumer will not ask for these messages to the *MessageProducer*.

7.5.2 How To Configure a MessageConsumer for Durable Consumer State

To configure a *MessageConsumer* with durable consumer state, use the property QoS policy associated with a *MessageConsumer* and/or *ConnectionFactory*. A property defined in the *ConnectionFactory* will be applicable to all the *MessageConsumers* contained in *Connections* created by that factory unless it is overwritten by a *MessageConsumer*.

The property QoS policy is made up of key-value properties, and is configured as follows:

```
<property>
    <value>
            <element>
                    <name>property1</name>
                    <value>value1</value>
            </element>
            <element>
                    <name>property2</name>
                    <value>value2</value>
            </element>
    </value>
</property>
```

Table 7.2 lists the supported properties.

Table 7.2 **Durable Consumer State Properties**

Property	Description
dds.data_reader.state. odbc. dsn	The ODBC DSN (Data Source Name) associated with the database where the *MessageConsumer* state must be persisted. This property is required.

Table 7.2 **Durable Consumer State Properties**

Property	Description
dds.data_reader.state. filter_redundant_messages	To enable durable consumer state, this property must be set to "1". Otherwise, the consumer state will not be kept and/or persisted. When the consumer state is not maintained, *RTI Message Service* does not filter duplicate messages that may be coming from the same virtual producer. By default, this property is set to "1" when either of these are true: ❑ The *MessageConsumer*'s durability **kind** is PERSISTENT_DURABILITY_QOS or TRANSIENT_DURABILITY_QOS ❑ **dds.data_reader.state.odbc.dsn** is configured
dds.data_reader.state. odbc. driver	This property indicates which ODBC driver to load. If the property is not specified, *RTI Message Service* will try to use the standard ODBC driver manager library (UnixOdbc on UNIX/Linux systems, the Windows ODBC driver manager on Windows systems).
dds.data_reader.state.odbc. username	These two properties configure the username and password used to connect to the database.
dds.data_reader.state.odbc. password	
dds.data_reader.state.restore	This property indicates if the persisted *MessageConsumer* state must be restored or not once the *MessageConsumer* is restarted. If this property is 0, the previous state will be deleted from the database. If it is 1, the *MessageConsumer* will restore its previous state from the database content.
dds.data_reader.state. checkpoint_frequency	This property controls how often the consumer state is stored into the database. A value of *N* means store the state once every *N* messages. A high frequency will provide better performance. However, if the consumer is restarted it may receive some duplicate messages. These messages will be filtered by *RTI Message Service* and they will not be propagated to the application.

Table 7.2 **Durable Consumer State Properties**

Property	Description
dds.data_reader.state. persistence_service. request_depth	This property indicates how many of the most recent historical messages the persisted *MessageConsumer* wants to receive upon start-up.

Example Configuration:

```
<topic name="Example Topic">
    <consumer_defaults>
         <protocol>
              <virtual_guid>
                   <value>
  239,238,237,236,235,234,233,232,231,231,229,228,227,226,225,223
                   </value>
              </virtual_guid>
         </protocol>
         <property>
              <value>
                   <element>
                       <name>dds.data_consumer.state.odbc.dsn</name>
                       <value>my user DSN</value>
                   </element>
                   <element>
                       <name>dds.data_consumer.state.odbc.driver
                       </name>
                       <value>my ODBC library</value>
                   </element>
                   <element>
                       <name>dds.data_consumer.state.restore</name>
                       <value>1</value>
                   </element>
              </value>
         </property>
    </consumer_defaults>
</topic>
```

Chapter 8 Scalable High-Performance Applications: Keys

To build scalable, high-performance applications, it's not enough to simply be familiar with **send** and **receive** methods. You need to understand the requirements of your data stream and know how to declare those requirements to the middleware so that it can deliver to you the semantics and performance you need and expect.

This chapter describes how you can improve scalability and simplify configuration management by distinguishing among subsets of a topic's message stream using a concept called *keys*. It assumes that you have already read the Getting Started Guide and executed the tutorial there. It further assumes that you have read and understand the more-general information contained in the earlier chapters of this manual, including Chapter 6: Scalable High-Performance Applications: Message Reliability and Chapter 7: Scalable High-Performance Applications: Durability and Persistence for High Availability.

This chapter is organized as follows:

8.1 Introduction to Keys

A topic defines an extensive set of QoS that govern the communication between the publishers and subscribers that use that topic. The application- and system-level requirements that those QoS enforce are generally common to a large number of similar objects in the world, to which middleware messages pertain.

For example, a market data distribution system may disseminate information about many different stocks. However, most of the QoS pertaining to that stock data are the same, regardless of that the particular stock is: the level of reliability required, the amount of historical data that will be kept and on what basis, and so forth. It is very unlikely that the market data infrastructure will want to distribute Apple and Intel data reliably but Microsoft and IBM data in a best-effort fashion, for instance.

For example, a radar track management system may track very many objects. These objects have distinct identities, but the system will track them all in a consistent way. And unlike stocks, the set of objects tracked by a radar is open; it is not possible to know ahead of time all of the objects that might be detected. New objects can appear at any time, and the same object can come and go quickly and repeatedly. Setting up and tearing down topics and the message producers and consumers that use them very rapidly is not an efficient use of CPU and network resources.

For these reasons, it is often best to consider a topic to be a relatively coarse-grained communication pathway. Using separate topics for fine-grained information—such as for each stock or each flight detected by a radar—can lead to heavy resource usage and duplicated configuration data.

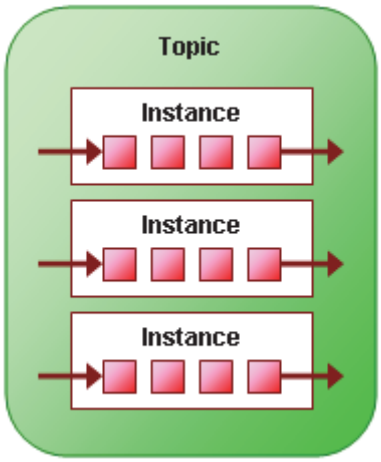
Although different real-world objects may be governed by the same set of requirements, it is desirable for the middleware to distinguish among them. Returning to the market data distribution system example, there may be a requirement to maintain the latest price at all times so that new subscribers can access it immediately. But this price should be maintained *separately for each stock*; a single last price, which could refer to Apple at one moment and IBM the next, would not be very useful.

Relational databases addresses this critical use case with a concept called a *key*: records pertaining to multiple objects of the same type can be stored in the same table; a field called the "key" distinguishes them from one another. *RTI Message Service* takes the same approach.

In *RTI Message Service*, the key is a well-known string property in a message. If no value has been set, the empty string is assumed.

```
myMessage.setStringProperty("JMS_RTIKey", "the key value");
```

The messages that share a key value are referred to as an *instance*.



To enable per-instance message management, a topic must be marked as *keyed* in its configuration file:

```
<topic name="SampleTopic">
    <keyed>true</keyed>
    <!-- ... -->
</topic>
```

If the **<keyed></keyed>** property is not specified, it is assumed to have the value *false*.

8.2 QoS Configuration

Turning on key support allows several QoS policies already described in other chapters to operate on a per-instance basis.

- The history depth that defines (*a*) the space window for reliability and (*b*) the number of historical messages to be provided to late-joining subscribers is enforced per-instance. For example, if the depth is 5, the most-recent 5 messages *from each instance* will be retained by the middleware.

- ❏ The resource limits QoS policy defines additional fields that allow applications to both manage the overall memory usage available to keyed topics as well as to implement fairness policies to ensure that more-frequently-updated instances do not displace data from less-frequently-updates ones.

The history, reliability, and resource limits QoS policies are described generally—without a treatment of keys—in Chapter 6: Scalable High-Performance Applications: Message Reliability. Additional information about those policies in the context of message persistence is given in Chapter 7: Scalable High-Performance Applications: Durability and Persistence for High Availability.

Example: Air Traffic Control

Consider an application that publishes altitude and velocity information for aircraft. These aircraft are distinguished by their flight numbers.

The configuration file specifies that updates should be propagated reliably, but that only the latest state of each flight should be retained.

```
<topic name="Example Topic">
    <keyed>true</keyed>
    <reliability>
           <kind>RELIABLE_RELIABILITY_QOS</kind>
    </reliability>
    <history>
           <!-- May be omitted: <kind>KEEP_LAST_HISTORY_QOS</kind> -
->
           <depth>1</depth>
    </history>
    <durability>
           <kind>TRANSIENT_LOCAL_DURABILITY_QOS</kind>
    </durability>
</topic>
```

The application code uses the **JMS_RTIKey** property to distinguish among messages.

```
MapMessage myFlight = mySession.createMapMessage();
myFlight.setStringProperty("JMS_RTIKey", "NW123");
myFlight.setInt("AltitudeInFt", 30000);
myFlight.setDouble("SpeedInMph", 400.0);
myProducer.send(myFlight);
```

8.2.1 Fairness and Resource Management

The resource limits QoS policy defines two fields that applications can use to manage instance resource consumption:

- The **max_instances** field controls the total number of instances about which the messaging infrastructure will retain data. By default, this value is unlimited, allowing the middleware to allocate memory as needed to work with any number of instances.
- The **max_messages_per_instance** allows the application to control resource allocation among instance. By default, this value is unlimited, allowing messages to be processed without regard to their instance. If this value is set to a finite value (which must be less than or equal to **max_messages**), it can prevent more-frequently-updated instances from consuming all memory resources and starving out less-frequently-updates instances. This scenario is not a concern when resource usage is not limited (*i.e.* when **max_messages** has its default, unlimited, value), but can be important when overall resource usage is limited.

When a message producer exceeds one of these limits while sending a new message, it will block until it receives sufficient acknowledgements to make space available in its history cache.

When a message consumer exceeds one of these limits while reading a message from the network, it will discard the message without acknowledgement and update its **MESSAGE_REJECTED** status. As the subscribing application processes its backlog of received messages, making space in the writer's queue available, the producer will resend the previously-rejected messages if they are still available. See Chapter 6: Scalable High-Performance Applications: Message Reliability for more information about the **MESSAGE_REJECTED** status.

Example: Resource Sharing

In this example, both publisher and subscriber are permitted to store only 25 historical messages of up to 25 instances. However, any single instance is allowed to use no more than 10 of those 25 "slots."

```
<topic name="Example Topic">
    <reliability>
           <kind>RELIABLE_RELIABILITY_QOS</kind>
    </reliability>
    <history>
           <kind>KEEP_ALL_HISTORY_QOS</kind>
    </history>
```

```
    <resource_limits>
           <max_messages>25</max_messages>
           <max_instances>25</max_instances>
           <max_messages_per_instance>10</max_messages_per_instance>
    </resource_limits>
</topic>
```

8.3 Debugging Configuration Problems: Inconsistent Topic Notifications

If your configuration management procedures break down, you may find that one application in your distributed system has configured the topic of a given name as keyed and another has not.

Application Program A:

```
<topic name="SampleTopic">
    <keyed>true</keyed>
    <!-- ... -->
</topic>
```

Application Program B:

```
<topic name="SampleTopic">
    <keyed>false</keyed>
    <!-- ... -->
</topic>
```

The **StatusNotifier** class described in Chapter 2: Connecting to the Network provides synchronous and asynchronous notifications of just this event.

Table 8.1 **Notification Type: StatusNotifier.INCONSISTENT_TOPIC_NOTIFICATION_TYPE**

The middleware has detected that another application in the distributed system has started publishing or subscribing to a topic of the same name as a topic used by this session but having a different key configuration.		
Attribute Name	**Attribute Type**	**Description**
totalCount	int	The total number of inconsistent topic definitions discovered in the distributed system.
totalCountChange	int	The number of inconsistent topic definitions discovered in the distributed system since the last time this status was queried. If your application receives status notifications via a listener callback, this number will generally be 1. If your application polls for status changes, it may be zero or more.

Example:

```
Session mySession = …;
Topic myTopic = …;

StatusNotifier myNotifier = new StatusNotifier(mySession)
Status inconsistencies = myNotifier.getStatus(
    myTopic,
    StatusNotifier.INCONSISTENT_TOPIC_NOTIFICATION_TYPE);

int numInconsistencies = inconsistencies.getIntAttribute(
    "totalCount");
```

Appendix A JMS Conformance

RTI Message Service is a high-performance peer-to-peer messaging system designed from the ground up for demanding low-latency, real-time systems. The JMS specification, in contrast, was developed by vendors of more traditional brokered enterprise messaging systems. Because of these different histories, not all elements of the JMS specification are appropriate, and not all capabilities of *RTI Message Service* have analogues in JMS. This chapter summarizes those areas of the JMS specification that are not supported, as well as some of the unique capabilities of *RTI Message Service*.

This appendix is organized as follows:

A.1 Message Filtering

RTI Message Service supports two broad classes of message filtering:

- **Time-based filters**, specified in a configuration file. This capability helps relatively slow consumers from being overwhelmed by high-rate periodic data from faster producers. It is an extension to the JMS specification.
- **Content-based filters**, specified with a message selector when creating a *MessageConsumer*. This capability reduces the application-level filtering burden by ensuring that each consumer delivers only messages that are of interest.

RTI extends the standard SQL grammar with a new keyword **MATCH**, which compares message fields against dot-delimited regular expressions. This feature eases migration from TIBCO Rendezvous.

However, *RTI Message Service* supports filters only on a well-defined subset of the possible message header fields and properties:

- **JMS_RTIKey** property
- **JMSCorrelationID** header
- **JMSReplyTo** header
- **JMSType** header

A.2 Message Durability and Persistence

The JMS specification supports two mechanisms for persisting sent messages:

- Persistent message producers (**DeliveryMode.PERSISTENT**) persist all sent messages to permanent storage to increase message availability in case the producer fails and must be restarted. However, if a vendor-specific retention policy on the subscriber does not provide for a message's delivery, it could still be lost.
- The durable subscription feature (**Session.createDurableSubscriber**) allows named subscriptions to be started, stopped, and moved from node to node without any associated message loss. However, until the subscription has been created for the first time, no messages will be retained.

These capabilities, while valuable, are problematic when applied to low-latency real-time systems, largely because of poor integration between them, underspecified behavior, and poorly chosen defaults.

- The default delivery mode (persistent) is the mode with the highest latency, highest jitter, and highest cost of deployment, because it requires the management of an independent persistence mechanism.
- There is no specified way to describe how much data should be persisted, for how long, under what conditions.
- There is no specified mechanism for maintaining and providing historical data for late-joining subscribers.

RTI Message Service provides a comprehensive model for message durability and persistence that incorporates both publication and subscription roles in a consistent way. The following durability kinds are supported on both publisher and subscriber:

- **VOLATILE**—The middleware does not need to keep any message on behalf of any subscriber that is unknown to the publisher at the time the message is written. Once a message has been acknowledged by all known subscribers, it can be removed from the publisher's cache. If the publisher fails before the message is acknowledged, the message could be lost. *This is the highest performing configuration and the default.*
- **TRANSIENT_LOCAL**—The publisher will keep some number of historical messages so that they can be delivered to any potential late-joining subscribers. However, the messages will not be persisted to permanent storage, so if the publisher fails before the message is acknowledged, the message could be lost.
- **TRANSIENT**—The publisher will keep some number of historical messages so that they can be delivered to any potential late-joining subscribers. These messages will be retained in a persistence service external to the original publisher, so that if the publisher fails, the message will remain available. However, the message may not be persisted to permanent storage, so if the publisher and the service both fail, the message could be lost.
- **PERSISTENT**—Messages are kept in permanent storage external to the original publisher so that they can outlive multiple failures or even a full system restart.

Related QoS policies govern the quantity of historical data retained, both in terms of a maximum number of messages and retention time. For more information about all of these policies and how to configure them, see Chapter 7: Scalable High-Performance Applications: Durability and Persistence for High Availability.

These QoS policies are specified in the configuration file loaded by the application. Once the *InitialContext* has been created, they cannot be changed. This is because the mapping between the JMS-specified delivery modes and *RTI Message Service* durability kinds is not one-to-one. When calling **MessageProducer.getDeliveryMode()**, the RTI PERSISTENT **kind** will be reported as the JMS PERSISTENT mode. All of RTI durability **kind** values will be reported as NON_PERSISTENT. However, setting a NON_PERSISTENT mode does not have a defined meaning.

Because the **noLocal** option configuration on message consumers is most used and useful when working with JMS durable subscribers, it is not supported; this argument, when provided, must be set to **false**.

A.3 Reliability and Acknowledgement

The JMS specification assumes that all message delivery is reliable, and describes three acknowledgement modes:

- **AUTO_ACKNOWLEDGE**—The middleware automatically acknowledges a message's receipt upon completion of a message listener, if any, or return from a **receive** call.
  - This model performs well in the case of asynchronous delivery with a message listener. However, the acknowledgement behavior is unspecified in the event that an exception is thrown from the listener.
  - In high-volume data flows, performance and determinism cannot be maintained without a message listener if the subscriber fails to call **receive** sufficiently often.
- **CLIENT_ACKNOWLEDGE**—The middleware acknowledges a consumed message by calling the message's acknowledge method.

  The lack of determinism makes this option inappropriate for high-performance or high-volume data streams.
- **DUPS_OK_ACKNOWLEDGE**—The middleware may lazily acknowledge the delivery of messages. This behavior may result in the delivery of some duplicate messages if there is a middleware failure.

Best-effort delivery—which is important for many sensor data, audio, video, and other periodic and/or streaming data applications—is not addressed by the specification. Furthermore, acknowledgement is configured at the *Session* level, not the data stream

(*Topic*) level, leaving unspecified the behavior when applications with different acknowledgement expectations attempt to communicate.

RTI Message Service operates in one of two reliability modes: RELIABLE or BEST_EFFORT. Both are specified per-*Topic* in the configuration file; the AUTO_ACKNOWLEDGE and DUPS_OK_ACKNOWLEDGE acknowledgement kinds are accepted in application code, although they are otherwise ignored. Best-effort mode is the most deterministic and the default if nothing is specified in the file.

When a *Topic* is configured for reliable communication, the acknowledgement is sent automatically after the message listener, if any, returns from its **onMessage** call; this is consistent with the AUTO_ACKNOWLEDGE mode. However, for the sake of performance when a message listener is not installed, acknowledgements are sent as soon as a message is available to be received. (That is, the message has arrived and has been properly ordered in the message stream.) The middleware does not wait for **receive** to actually be called by the application.

Provided that the subscriber does not fail and restart, applications will never observe duplicate messages. To avoid duplicate messages, you can configure your subscriber to persist its acknowledgement state such that if it restarts, it will not deliver any duplicate messages.

For more information about reliability and durability, see Chapter 6: Scalable High-Performance Applications: Message Reliability.

A.4 Transaction Support

The JMS specification allows sessions to be created in a transacted mode, such that all send and receive actions that take place in between **commit()** calls are considered atomic operations. This behavior is not supported.

In its place, *RTI Message Service* supports an alternative transaction mechanism. Sequences of messages sent from a single session can be grouped into a "coherent set," such that they will be received atomically by any subscribing application: either all messages in the set will be made available to the application or none of them will. This model is more flexible than the JMS-specified model, because it allows a single *Session* object to operate in either a transacted or non-transacted mode as required. However, it does not allow any message acknowledgements to be cancelled.

For more information about RTI's transaction support, see Coherent Changes (Section 4.2.4).

A.5 Message Queue Support

RTI Message Service is a publish/subscribe middleware. It does not support message queues. Specifically, the following APIs are not implemented:

- *Queue* and *QueueBrowser* interfaces
- The *Session's* **createBrowser(), createQueue(),** and **createTemporaryQueue ()** methods

Applications that require message queues can use *RTI Message Service* alongside a second JMS implementation that does support them. Your application can configure its JNDI repository with an *RTI Message Service* configuration file, from which it can look up *Topic* and *TopicConnectionFactory* objects, and a second directory service, from which it can look up *Queue* and *QueueConnectionFactory* objects.

A.6 Message Producer Configuration

In *RTI Message Service*, QoS is carefully and deterministically configured and managed on a per-topic, per-message producer/consumer collaboration basis. Furthermore, resource allocation is put strictly under the control of the application.

Some message producer methods are difficult to reconcile with this model, and consequently there are some caveats in the support for these methods.

Note: The maximum message size that can be sent through the JMS API is limited to 63KB (even when using a transport that supports larger message sizes, such as shared memory or TCP).

A.6.1 Producer Priority

Message producers will preserve any priority value set by the application, but these values will have no effect.

To minimize latency, in publishing application, *RTI Message Service* publishes messages onto the network in order, usually within the context of the **publish** or **send** call. The priority is irrelevant: messages will be sent immediately regardless.

A similar situation applies in a subscribing application. Provided that messages arrive in order, they will be delivered to the application immediately as they are read from the socket. No intermediate queuing is necessary, and hence no prioritization.

A.6.2 Per-Message Destinations

The JMS specification states that a *MessageProducer* can be created either (*a*) with a destination (topic), to which it will publish all messages, or (*b*) without a topic; the topic will be specified separately with each **send** or **publish** call.

This model is difficult to reconcile with the strict QoS constraints and deterministic resource management offered by *RTI Message Service*, because the **send** and **publish** methods that accept a destination argument provide no way to deterministically control when the producer-side message caches are initialized or what QoS the producer should use. Consequently, these methods are only supported when a message producer on the given destination has already been created in the current session; they are not supported for arbitrary destinations.

Example:

```
Topic topic1 = …;
Topic topic2 = …;
Topic topic3 = …;
Session mySession = …;

MessageProducer producer1 = mySession.createProducer(topic1);
MessageProducer producer2 = mySession.createProducer(topic2);
Message message = mySession.createMessage();
producer1.send(topic2, message);// supported
producer2.send(topic1, message);// supported
producer1.send(topic3, message);// NOT SUPPORTED: throws JMSExcep-
tion
```

In the last line, the QoS of the data stream of which the message would be a part is ill-defined.

A.6.3 Per-Message QoS Configuration

The *MessageProducer* interface provides **send** and **publish** methods that accept message-specific QoS values: **deliveryMode**, **priority**, and **timeToLive.** These arguments have limited applicability:

- ❏ The delivery mode must match the current delivery mode of the message producer; per-message changes are not supported. For more information about the persistence model supported by *RTI Message Service*, see Message Durability and Persistence (Section A.2).
- ❏ Any valid priority value is accepted but will be ignored; see Producer Priority (Section A.6.1).

The middleware implements the time-to-live QoS on a per-producer basis, not a per-message basis. If a different time to live is specified for a particular message, the middleware will modify the producer's configuration, send the message, and then restore the producer's configuration. However, object configuration data is propagated on the network separately from application data, so whether consumers process this configuration change correctly depends on the order in which they process data from these two data streams.

For these reasons, RTI strongly advises that applications do not use per-message QoS values.

A.7 Optional JMS Methods

Some methods in the JMS API are optional parts of the specification. *RTI Message Service* does not implement all of these. Specifically, it does not support the following methods:

- ❏ **Connection.createConnectionConsumer**
- ❏ **Connection.createDurableConnectionConsumer**
- ❏ **Session.run**
- ❏ **Session.get/setMessageListener**[1]

1. This listener is referred to as the "distinguished message listener," and represents an expert facility not used by most JMS applications. The more conventional practice of installing a message listener with a *MessageConsumer* is fully supported—and indeed recommended to the lowest possible latency.